

elixir

目錄

Elixir 程序设计

Elixir编程入门

> 作者：straightdave
>
> 来源：programming_elixir

Elixir，[ɪ'lɪksər]，意为灵丹妙药、圣水，其logo是一枚紫色水滴：



Elixir是一门建立在Erlang虚拟机上的函数式的系统编程语言，支持元编程。创始人José Valim是ruby界的知名人士。
私以为，可以把Elixir看作函数式的ruby语言，或者是语法类似ruby的Erlang。Elixir受瞩目的原因，是因为它结合了Erlang作为系统编程语言的各种优点，以及ruby那样简单易懂的语法（Erlang语法比较晦涩）。

Elixir还是一门初出茅庐的语言：
2014年8月31日，1.0.0发布
2014年9月1日临晨，1.0.0rc1发布
2014年9月7日晚，1.0.0rc2发布
2014年9月10日，1.0.0正式发布
2015年9月28日，1.1发布
2016年1月1日，v1.2.0发布

本文主要框架为Elixir官方的入门教程，辅以网上其它Elixir资源的内容，以及花钱:sob:购买的原版书籍（Dave Thomas的《Programming Elixir》，Progmatic）

> 请帮助更新文档(pr)。有问题请发issue

基本教程

- 1-简介
- 2-基本数据类型
- 3-基本运算符
- 4-模式匹配
- 5-流程控制
- 6-二进制-字符串-字符列表
- 7-键值-图-字典
- 8-模块
- 9-递归

1-简介

欢迎！

> 本章是主要讲了各个平台上如何安装使用Elixir。由于本文主要关注Elixir的语言学习， 因此这个章节所讲的步骤或工具可能不是最新，请大家自行网上搜索。

本章将涵盖如何安装Elixir，并且学习使用交互式的Elixir Shell（称为IEx）。

使用本教程的需求：

- Erlang - version 17.0 或更高
- Elixir - 1.0.0 或更高

现在开始吧！

> 如果你发现本手册有错误，请帮忙开*issue*讨论或发*pull request*。

1.1-安装包

在各个平台上最方便的安装方式是相应平台的安装包。 如果没有，推荐使用precompiled package或者用源码编译安装。

注意Elixir需要Erlang 17.0或更高。下面介绍的方法基本上都会自动为你安装Erlang。 假如没有，请阅读下面安装Erlang的说明。

Mac OS X

- Homebrew
  - 升级Homebrew到最新
  - 执行：`brew install elixir`
- Macports
  - 执行：`sudo port install elixir`

Unix（或者类Unix）

- Fedora 17或更新
  - 执行：`yum install elixir`
- Fedora 22或更新
  - 执行：`dnf install elixir`
- Arch Linux (社区repo)
  - 执行：`pacman -S elixir`

- openSUSE (and SLES 11 SP3+)
  - 添加Erlang devel repo: `zypper ar -f obs://devel:languages:erlang/ erlang`
  - 执行：`zypper in elixir`
- Gentoo
  - 执行：`emerge --ask dev-lang/elixir`
- FreeBSD
  - 使用ports: `cd /usr/ports/lang/elixir && make install clean`
  - 或使用pkg: `pkg install elixir`
- Ubuntu 12.04和14.04，或Debian 7
  - 添加Erlang Solutions repo:
    `wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb && sudo dp`
  - 执行：`sudo apt-get update`
  - 安装Erlang/OTP平台及相关程序：`sudo apt-get install esl-erlang`
  - 安装Elixir：`sudo apt-get install elixir`

Windows

- Web installer
  - 下载installer
  - 点下一步，下一步...直到完成
- Chocolatey
  - `cinst elixir`

1.3-使用预编译包

Elixir为每一个release提供了预编译包（编译好并打包的程序，开箱即用）。
首先安装Erlang， 然后在这里下载最新的 预编译包（Precompiled.zip）。unzip，即可使用elixir程序和iex程序了。
当然为了方便起见，需要将一些路径加入环境变量。

1.4-从源码编译安装（Unix和MinGW）

首先安装Erlang， 然后在这里下载最新的源码， 自己使用make工具编译安装。

> 在Windows上编译安装请参考https://github.com/elixir-lang/elixir/wiki/Windows
>
> 附上加环境变量的命令
>
> ```
> $ export PATH="$PATH:/path/to/elixir/bin"
> ```

如果你十分激进，可以直接选择编译安装github上的master分支：

```
$ git clone https://github.com/elixir-lang/elixir.git
$ cd elixir
$ make clean test
```

如果测试无法通过，可在repo里开issue汇报。

1.5-安装Erlang

安装Elixir唯一的要求就是Erlang（V17.0+）， 它可以很容易地使用 预编译包安装。 如果你想从源码安装，可以去Erlang网站找找， 参考Riak文档。
安装好Erlang后，打开命令行（或命令窗口），输入 `erl` ，可以输出Erlang的版本信息：

```
Erlang/OTP 17 (erts-6) [64-bit] [smp:2:2] [async-threads:0] [hipe] [kernel-poll:false]
```

> 安装好Erlang后，你需要手动添加环境变量或$PATH。 关于环境变量，参考这里。

1.6-交互模式

安装好Elixir之后，你有了三个可执行文件： `iex` ， `elixir` 和 `elixirc` 。 如果你是用预编译包方式安装的，可以在解压后的bin目录下找到它们。

现在我们可以从 `iex` 开始了（或者是 `iex.bat` ，如果在Windows上）。 交互模式，就是可以向其中输入任何Elixir表达式或命令，然后直接看到表达式或命令的结果。 如以下所示：

```
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)

iex> 40 + 2
42
iex> "hello" <> " world"
"hello world"
```

对这种交互式命令行，相信熟悉ruby，python等动态语言的程序员一定不会陌生。

> 如果你用的是Windows，你可以使用 `iex.bat --werl` ，可以根据你的console获得更好的使用体验。

1.7-执行脚本

把表达式写进脚本文件，可以用 `elixir` 命令执行它。如：

```
$ cat simple.exs
IO.puts "Hello world
from Elixir"

$ elixir simple.exs
Hello world
from Elixir
```

在以后的章节中，我们还会介绍如何编译Elixir程序，以及使用Mix这样的构建工具。

2-基本类型

本章介绍Elixir的基本类型。Elixir主要的基本类型有：整型（integer），浮点（float），布尔（boolean），原子（atom，又称symbol符号），字符串（string），列表（list）和元组（tuple）等。

它们在iex中显示如下：

```
iex> 1          # integer
iex> 0x1F       # integer
iex> 1.0        # float
iex> true       # boolean
iex> :atom      # atom / symbol
iex> "elixir"   # string
iex> [1, 2, 3]  # list
iex> {1, 2, 3}  # tuple
```

2.1-基本算数运算

打开`iex`，输入以下表达式：

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
```

> `10 / 2`返回了一个浮点型的5.0而非整型的5，这是预期的。
> 在Elixir中，`/`运算符总是返回浮点型数值。

如果你想进行整型除法，或者求余数，可以使用函数`div`和`rem`。（rem的意思是division remainder，余数）：

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

> 在写函数参数时，括号是可选的。（ruby程序员会心一笑）

Elixir支持用 捷径（**shortcut**） 书写二进制、八进制、十六进制整数。如：

```
iex> 0b1010
10
iex> 0o777
511
iex> 0x1F
31
```

> 揉揉眼，八进制是 `0o` ，数字0 + 小写o。

输入浮点型数字需要一个小数点，且在其后至少有一位数字。 Elixir支持使用 `e` 来表示指数：

```
iex> 1.0
1.0
iex> 1.0e-10
1.0e-10
```

Elixir中浮点型都是64位双精度。

2.2-布尔

Elixir使用 `true` 和 `false` 两个布尔值。

```
iex> true
true
iex> true == false
false
```

Elixir提供了许多用以判断类型的函数，如 `is_boolean/1` 函数可以用来检查参数是不是布尔型。

> 在Elixir中，函数通过名称和参数个数（又称元数，arity）来识别。 如 `is_boolean/1` 表示名为 `is_boolean` ，接受一个参数的函数； 而 `is_boolean/2` 表示与其同名、但接受2个参数的不同函数。（只是打个比方，这样的is_boolean实际上不存在）
> 另外， `<函数名>/<元数>` 这样的表述是为了在讲述函数时方便，在实际程序中如果调用函数， 是不用注明 `/1` 或 `/2` 的。

```
iex> is_boolean(true)
true
iex> is_boolean(1)
false
```

类似的函数还有 `is_integer/1` ， `is_float/1` ， `is_number/1` ， 分别测试参数是否是整型、浮点型或者两者其一。

> 可以在交互式命令行中使用 `h` 命令来打印函数或运算符的帮助信息。
> 如 `h is_boolean/1` 或 `h ==/2` 。 注意此处提及某个函数时，不但要给出名称，还要加上元数 `/<arity>` 。

2.3-原子

原子（atom）是一种常量，名字就是它的值。有些语言中称其为 符号（**symbol**）（如ruby）：

```
iex> :hello
:hello
iex> :hello == :world
false
```

布尔值 `true` 和 `false` 实际上就是原子：

```
iex> true == :true
true
iex> is_atom(false)
true
```

2.4-字符串

在Elixir中，字符串以 双括号 包裹，采用UTF-8编码：

```
iex> "hellö"
"hellö"
```

Elixir支持字符串插值（和ruby一样使用 `#{ ... }`）：

```
iex> "hellö #{:world}"
"hellö world"
```

字符串可以直接包含换行符，或者其转义字符：

```
iex> "hello
...> world"
"hello\nworld"
iex> "hello\nworld"
"hello\nworld"
```

你可以使用 `IO` 模块（module）里的 `IO.puts/1` 方法打印字符串：

```
iex> IO.puts "hello\nworld"
hello
world
:ok
```

函数 `IO.puts/1` 打印完字符串后，返回原子值 `:ok` 。

字符串在Elixir内部被表示为二进制数值（binaries），也就是一连串的字节（bytes）：

```
iex> is_binary("hellö")
true
```

> 注意，二进制数值（binary）是Elixir内部的存储结构之一。 字符串、列表等类型在语言内部就表示为二进制数值，因此它们也可以被专门操作二进制数值的函数修改。

你可以查看字符串包含的字节数量：

```
iex> byte_size("hellö")
6
```

> 为啥是6？不是5个字符么？注意里面有一个非ASCII字符 `ö` ，在UTF-8下被编码为2个字节。

我们可以使用专门的函数来返回字符串中的字符数量：

```
iex> String.length("hellö")
5
```

String模块中提供了 很多符合Unicode标准的函数来操作字符串。如：

```
iex> String.upcase("hellö")
"HELLÖ"
```

记住，单引号和双引号包裹的字符串在Elixir中是两种不同的数据类型：

```
iex> 'hellö' == "hellö"
false
```

我们将在之后关于“二进制、字符串与字符列表”章节中详细讲述它们的区别。

2.5-匿名函数

在Elixir中，使用关键字 `fn` 和 `end` 来界定函数。如：

```
iex> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> is_function(add)
true
iex> is_function(add, 2)
true
iex> is_function(add, 1)
false
iex> add.(1, 2)
3
```

在Elixir中，函数是 一等公民。你可以将函数作为参数传递给其他函数，就像整型和浮点型一样。 在上面的例子中，我们向函数 `is_function/1` 传递了由变量 `add` 表示的匿名函数， 结果返回 `true` 。 我们还可以调用函数 `is_function/2` 来判断该参数函数的元数（参数个数）。

注意，在调用一个匿名函数时，在变量名和写参数的括号之间要有个 点号**(.)**。

匿名函数是闭包，意味着它们可以保留其定义的作用域（scope）内的其它变量值：

```
iex> add_two = fn a -> add.(a, 2) end
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> add_two.(2)
4
```

这个例子定义的匿名函数 `add_two` 它内部使用了之前在同一个iex内定义好的 `add` 变量。 但要注意，在匿名函数内修改了所引用的外部变量的值，并不实际反映到该变量上：

```
iex> x = 42
42
iex> (fn -> x = 0 end).()
0
iex> x
42
```

这个例子中匿名函数把引用了外部变量x，并修改它的值为0。这时函数执行后，外部的x没有被影响。

2.6-（链式）列表

Elixir使用方括号标识列表。列表可以包含任意类型的值：

```
iex> [1, 2, true, 3]
[1, 2, true, 3]
iex> length [1, 2, 3]
3
```

两个列表可以使用 `++/2` 拼接，使用 `--/2` 做“减法”：

```
iex> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex> [1, true, 2, false, 3, true] -- [true, false]
[1, 2, 3, true]
```

本教程将多次涉及列表头（head）和尾（tail）的概念。 列表的头指的是第一个元素，而尾指的是除了第一个元素以外，其它元素组成的列表。 它们分别可以用函数 `hd/1` 和 `tl/1` 从原列表中取出：

```
iex> list = [1,2,3]
iex> hd(list)
1
iex> tl(list)
[2, 3]
```

尝试从一个空列表中取出头或尾将会报错：

```
iex> hd []
** (ArgumentError) argument error
```

2.7-元组

Elixir使用大括号（花括号）定义元组（tuples）。类似列表，元组也可以承载任意类型的数据：

```
iex> {:ok, "hello"}
{:ok, "hello"}
iex> tuple_size {:ok, "hello"}
2
```

元组使用 连续的内存空间 存储数据。 这意味着可以很方便地使用索引访问元组数据，以及获取元组大小（索引从0开始）：

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
iex> tuple_size(tuple)
2
```

也可以很方便地使用函数 `put_elem/3` 设置某个位置的元素值：

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> put_elem(tuple, 1, "world")
{:ok, "world"}
iex> tuple
{:ok, "hello"}
```

注意函数 `put_elem/3` 返回一个新元组。原来那个由变量tuple标识的元组没有被改变。 这是因为Elixir的数据类型是 不可变的。 这种不可变性使你永远不用担心你的数据会在某处被某些代码改变。 在处理并发程序时，这种不可变性有利于减少多个程序实体同时修改一个数据结构时引起的竞争以及其他麻烦。

2.8-列表还是元组？

列表与元组的区别：列表在内存中是以链表的形式存储的，一个元素指向下一个元素， 然后再下一个...直到到达列表末尾。 我们称这样的一对数据（元素值 和 指向下一个元素的指针）为列表的一个单元（cons cell）。

用Elixir语法表示这种模式：

```
iex> list = [1|[2|[3|[]]]]
[1, 2, 3]
```

> 列表方括号中的竖线（|）表示列表头与尾的分界。

这个原理意味着获取列表的长度是一个线性操作：我们必须遍历完整个列表才能知道它的长度。 但是列表的前置拼接操作很快捷：

```
iex> [0] ++ list
[0, 1, 2, 3]
iex> list ++ [4]
[1, 2, 3, 4]
```

上面例子中第一条语句是 前置 拼接操作，执行起来很快。 因为它只是简单地添加了一个新列表单元，它的尾指针指向原先列表头部。而原先的列表没有任何变化。

第二条语句是 后缀 拼接操作，执行速度较慢。 这是因为它 重建 了原先的列表，让原先列表的末尾元素指向那个新元素。

另一方面，元组在内存中是连续存储的。 这意味着获取元组大小，或者使用索引访问元组元素的操作十分快速。 但是元组在修改或添加元素时开销很大，因为这些操作会在内存中对元组的进行整体复制。

这些讨论告诉我们当如何在不同的情况下选择使用不同的数据结构。

函数常用元组来返回多个信息。如 `File.read/1` ，它读取文件内容，返回一个元组：

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

如果传递给函数 `File.read/1` 的文件路径有效，那么函数返回一个元组， 其首元素是原子 `:ok` ，第二个元素是文件内容。 如果路径无效，函数也将返回一个元组，其首元素是原子 `:error` ，第二个元素是错误信息。

大多数情况下，Elixir会引导你做正确的事。 比如有个叫 `elem/2` 的函数，它使用索引来访问一个元组元素。 这个函数没有相应的列表版本，因为根据存储机制，列表不适用通过索引来访问：

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

当需要计算某数据结构包含的元素个数时，Elixir遵循一个简单的规则：如果操作在常数时间内完成（答案是提前算好的），这样的函数通常被命名为 `*size` 。而如果操作需要显式计数，那么该函数通常命名为 `*length` 。

例如，目前讲到过的4个计数函数：`byte_size/1`（用来计算字符串有多少字节），`tuple_size/1`（用来计算元组大小），`length/1`（计算列表长度）以及 `String.length/1`（计算字符串中的字符数）。

按照命名规则，当我们用 `byte_size` 获取字符串所占字节数时，开销较小。但是当我们用 `String.length` 获取字符串unicode字符个数时，需要遍历整个字符串，开销较大。

除了本章介绍的数据类型，Elixir还提供了 **Port**，**Reference** 和 **PID** 三个数据类型（它们常用于进程交互）。这些数据类型将在讲解进程时详细介绍。

3-基本运算符

通过前几章的学习，我们知道Elixir提供了 `+，-，*，/` 4个算术运算符，外加整数除法函数 `div/2` 和 取余函数 `rem/2` 。 Elixir还提供了 `++` 和 `--` 运算符来操作列表：

```
iex> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
iex> [1,2,3] -- [2]
[1,3]
```

使用 `<>` 进行字符串拼接：

```
iex> "foo" <> "bar"
"foobar"
```

Elixir还提供了三个布尔运算符： `or，and，not` 。这三个运算符只接受布尔值作为 第一个 参数：

```
iex> true and true
true
iex> false or is_atom(:example)
true
```

如果提供了非布尔值作为第一个参数，会报异常：

```
iex> 1 and true
** (ArgumentError) argument error
```

运算符 `or` 和 `and` 可短路，即它们仅在第一个参数无法决定整体结果的情况下才执行第二个参数：

```
iex> false and error("This error will never be raised")
false

iex> true or error("This error will never be raised")
true
```

> 如果你是Erlang程序员，Elixir中的 `and` 和 `or` 其实就是 `andalso` 和 `orelse` 运算符。

除了这几个布尔运算符，Elixir还提供 `||` ， `&&` 和 `!` 运算符。它们可以接受任意类型的参数值。 在使用这些运算符时，除了 false 和 nil 的值都被视作 true：

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
iex> !1
false
iex> !nil
true
```

根据经验，当参数确定是布尔时，使用 `and` ， `or` 和 `not` ； 如果非布尔值（或不确定是不是），用 `&&` ， `||` 和 `!` 。

Elixir还提供了 `==，!=，===，!==，<=，>=，<，>` 这些比较运算符：

```
iex> 1 == 1
true
iex> 1 != 2
true
iex> 1 < 2
true
```

其中 `==` 和 `===` 的不同之处是后者在判断数字时更严格：

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

在Elixir中，可以判断不同类型数据的大小：

```
iex> 1 < :atom
true
```

这很实用。排序算法不必担心如何处理不同类型的数据。总体上，不同类型的排序顺序是：

```
number < atom < reference < functions < port < pid < tuple < maps < list < bitstring
```

不用强记，只要知道有这么回事儿就可以。

4-模式匹配

本章起教程进入 不那么基础的 阶段，开始涉及函数式编程概念。 对之前没有函数式编程经验的人来说，这一章是一个基础，需要好好学习和理解。

在Elixir中，`=`运算符实际上叫做 匹配运算符。 本章将讲解如何使用`=`运算符来对各种数据结构进行模式匹配。 最后本章还会讲解pin运算符(`^`)，用来访问某变量之前绑定的值。

4.1-匹配运算符

我们已经多次使用`=`符号进行变量的赋值操作：

```
iex> x = 1
1
iex> x
1
```

在Elixir中，`=`作为 匹配运算符。下面来学习这样的概念：

```
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

注意`1 = x`是一个合法的表达式。 由于前面的例子给x赋值为1，因此在匹配时左右相同，所以它匹配成功了。而两侧不匹配的时候，MatchError将被抛出。

变量只有在匹配操作符`=`的左侧时才被赋值：

```
iex> 1 = unknown
** (RuntimeError) undefined function: unknown/0
```

错误原因是unknown变量没有被赋过值，Elixir猜你想调用一个名叫`unknown/0`的函数， 但是找不到这样的函数。

> 变量名在等号左边，Elixir认为是赋值表达式；变量名放在右边，Elixir认为是拿该变量的值和左边的值做匹配。

4.2-模式匹配

匹配运算符不光可以匹配简单数值，还能用来 解构 复杂的数据类型。例如，我们在元组上使用模式匹配：

```
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex> a
:hello
iex> b
"world"
```

在两端不匹配的情况下，模式匹配会失败。比方说，匹配的两端的元组不一样长：

```
iex> {a, b, c} = {:hello, "world"}
** (MatchError) no match of right hand side value: {:hello, "world"}
```

或者两端模式有区别（比如两端数据类型不同）：

```
iex> {a, b, c} = [:hello, "world", "!"]
** (MatchError) no match of right hand side value: [:hello, "world", "!"]
```

利用“匹配”的这个概念，我们可以匹配特定值，或者在匹配成功时。

下面例子中先写好了匹配的左端，它要求右端必须是个元组，且第一个元素是原子 `:ok` 。

```
iex> {:ok, result} = {:ok, 13}
{:ok, 13}
iex> result
13

iex> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
```

用在列表上：

```
iex> [a, 2, 3] = [1, 2, 3]
[1, 2, 3]
iex> a
1
```

列表支持匹配自己的 `head` 和 `tail` （这相当于同时调用 `hd/1` 和 `tl/1` ，给 `head` 和 `tail` 赋值）：

```
iex> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

同 `hd/1` 和 `tl/1` 函数一样，以上代码不能对空列表使用：

```
iex> [h|t] = []
** (MatchError) no match of right hand side value: []
```

> [head|tail]这种形式不光在模式匹配时可以用，还可以用作向列表插入前置数值：

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [0|list]
[0, 1, 2, 3]
```

模式匹配使得程序员可以容易地解构数据结构（如元组和列表）。 在后面我们还会看到，它是Elixir的一个基础，对其它数据结构同样适用，比如图和二进制。

小结：

- 模式匹配使用 `=` 符号
- 匹配中等号左右的“模式”必须相同
- 变量在等号左侧才会被赋值
- 变量在右侧时必须有值，Elixir拿这个值和左侧相应位置的元素做匹配

4.3-pin运算符

在Elixir中，变量可以被重新绑定：

```
iex> x = 1
1
iex> x = 2
2
```

> Elixir可以给变量重新绑定（赋值）。 它带来一个问题，就是对一个单独变量（而且是放在左端）做匹配时， Elixir会认为这是一个重新绑定（赋值）操作，而不会当成匹配，执行匹配逻辑。 这里就要用到pin运算符。

如果你不想这样，可以使用pin运算符(^)。 加上了pin运算符的变量，在匹配时使用的值是本次匹配前就赋予的值：

```
iex> x = 1
1
iex> ^x = 2
** (MatchError) no match of right hand side value: 2
iex> {x, ^x} = {2, 1}
{2, 1}
iex> x
2
```

注意如果一个变量在匹配中被引用超过一次，所有的引用都应该绑定同一个模式：

```
iex> {x, x} = {1, 1}
1
iex> {x, x} = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
```

有些时候，你并不在意模式匹配中的一些值。 可以把它们绑定到特殊的变量 “ _ ” (underscore)上。 例如，如果你只想要某列表的head，而不要tail值。你可以这么做：

```
iex> [h | _ ] = [1, 2, 3]
[1, 2, 3]
iex> h
1
```

变量“ _ ”特殊之处在于它不能被读，尝试读取它会报“未绑定的变量”错误：

```
iex> _
** (CompileError) iex:1: unbound variable _
```

尽管模式匹配看起来如此牛逼，但是语言还是对它的作用做了一些限制。 比如，你不能让函数调用作为模式匹配的左端。下面例子就是非法的：

```
iex> length([1,[2],3]) = 3
** (CompileError) iex:1: illegal pattern
```

模式匹配介绍完了。 在以后的章节中，模式匹配是常用的语法结构。

5-流程控制

本章讲解case，cond和if的流程控制结构。

5.1-case

case将一个值与许多模式进行匹配，直到找到一个匹配成功的：

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
```

如果与一个已赋值的变量做比较，要用pin运算符(^)标记该变量：

```
iex> x = 1
1
iex> case 10 do
...>   ^x -> "Won't match"
...>   _  -> "Will match"
...> end
```

可以加上卫兵子句（guard clauses）提供额外的条件：

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Won't match"
...> end
```

于是上面例子中，第一个待比较的模式多了一个条件：x必须是正数。

5.2-卫兵子句中的表达式

Erlang中只允许以下表达式出现在卫兵子句中：

- 比较运算符（==，!=，===，!==，>，<，<=，>=）
- 布尔运算符（and，or）以及否定运算符（not，!）
- 算数运算符（+，-，*，/）
- <>和++如果左端是字面值
- in运算符
- 以下类型判断函数：
  - is_atom/1
  - is_binary/1
  - is_bitstring/1
  - is_boolean/1
  - is_float/1
  - is_function/1
  - is_function/2
  - is_integer/1
  - is_list/1
  - is_map/1
  - is_number/1
  - is_pid/1
  - is_reference/1
  - is_tuple/1
- 外加以下函数：
  - abs(number)
  - bit_size(bitstring)
  - byte_size(bitstring)
  - div(integer, integer)
  - elem(tuple, n)
  - hd(list)
  - length(list)
  - map_size(map)
  - node()
  - node(pid | ref | port)
  - rem(integer, integer)
  - round(number)
  - self()
  - tl(list)
  - trunc(number)
  - tuple_size(tuple)

记住，卫兵子句中出现的错误不会漏出，只会简单地让卫兵条件失败：

```
iex> hd(1)
** (ArgumentError) argument error
    :erlang.hd(1)
iex> case 1 do
...>   x when hd(x) -> "Won't match"
...>   x -> "Got: #{x}"
...> end
"Got 1"
```

如果case中没有一条模式能匹配，会报错：

```
iex> case :ok do
...>   :error -> "Won't match"
...> end
** (CaseClauseError) no case clause matching: :ok
```

匿名函数也可以像下面这样，用多个模式或卫兵条件来灵活地匹配该函数的参数：

```
iex> f = fn
...>   x, y when x > 0 -> x + y
...>   x, y -> x * y
...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> f.(1, 3)
4
iex> f.(-1, 3)
-3
```

需要注意的是，所有case模式中表示的参数个数必须一致，否则会报错。 上面的例子两个待匹配模式都是x，y。如果再有一个模式表示的参数是x，y，z，那就不行：

```
iex(5)> f2 = fn
...(5)>   x,y -> x+y
...(5)>   x,y,z -> x+y+z
...(5)> end
** (CompileError) iex:5: cannot mix clauses with different arities in function definition
    (elixir) src/elixir_translator.erl:17: :elixir_translator.translate/2
```

5.3-cond

case是拿一个值去同多个值或模式进行匹配，匹配了就执行那个分支的语句。 然而，许多情况下我们要检查不同的条件，找到第一个结果为true的，执行它的分支。 这时我们用cond：

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

这样的写法和命令式语言里的 `else if` 差不多一个意思（尽管很少这么写）。

如果没有一个条件结果为true，会报错。因此，实际应用中通常会使用true作为最后一个条件。 因为即使上面的条件没有一个是true，那么该cond表达式至少还可以执行这最后一个分支：

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   true ->
...>     "This is always true (equivalent to else)"
...> end
```

用法就好像许多语言中，switch语句中的default一样。

最后需要注意的是，cond视所有非false和nil的值为true：

```
iex> cond do
...>   hd([1,2,3]) ->
...>     "1 is considered as true"
...> end
"1 is considered as true"
```

5.4 if和unless

除了case和cond，Elixir还提供了两很常用的宏： `if/2` 和 `unless/2` ， 用它们检查单个条件：

```
iex> if true do
...>   "This works!"
...> end
"This works!"
iex> unless true do
...>   "This will never be seen"
...> end
nil
```

如果给 `if/2` 的条件结果为false或者nil，那么它在do/end间的语句块就不会执行， 该表达式返回nil。 `unless/2` 相反。

它们都支持else语句块：

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"
```

> 有趣的是，`if/2`和`unless/2`是以宏的形式提供的，而不像在很多语言中那样是语句。可以阅读文档或`if/2`的源码（Kernel模块）。*Kernel* 模块还定义了诸如`+/2`运算符和`is_function/2`函数。它们默认被导入，因而在你的代码中可用。

5.5- `do` 语句块

以上讲解的4种流程控制结构：case，cond，if和unless，它们都被包裹在do/end语句块中。即使我们把if语句写成这样：

```
iex> if true, do: 1 + 2
3
```

在Elixir中，do/end语句块方便地将一组表达式传递给`do:`。以下是等同的：

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

我们称第二种语法使用了 关键字列表（**keyword lists**）。我们可以这样传递`else`：

```
iex> if false, do: :this, else: :that
:that
```

注意一点，do/end语句块永远是被绑定在最外层的函数调用上。例如：

```
iex> is_number if true do
...>  1 + 2
...> end
```

将被解析为：

```
iex> is_number(if true) do
...>  1 + 2
...> end
```

这使得Elixir认为你是要调用函数`is_number/2`（第一个参数是if true，第二个是语句块）。这时就需要加上括号解决二义性：

```
iex> is_number(if true do
...>  1 + 2
...> end)
true
```

关键字列表在Elixir语言中占有重要地位，在许多函数和宏中都有使用。后文中还会对其进行详解。

6-二进制-字符串-字符列表

UTF-8和Unicode
二进制（和bitstring）
字符列表

在“基本类型”一章中，介绍了字符串，以及使用 `is_binary/1` 函数检查它：

```
iex> string = "hello"
"hello"
iex> is_binary string
true
```

本章将学习理解，二进制（binaries）是个啥，它怎么和字符串扯上关系的。 以及用单引号包裹的值， `'like this'` 是啥意思。

6.1-UTF-8和Unicode

字符串是UTF-8编码的二进制。 为了弄清这句话啥意思，我们要先理解两个概念：bytes和code point的区别。 字母 `a` 的code point是97，而字母 `ł` 的code point是322。 当把字符串 `"hełło"` 写到硬盘上的时候，需要将其code point转化为bytes。 如果一个byte对应一个code point，那是写不了 `"hełło"` 的， 因为字母 `ł` 的code point是322，超过了一个byte所能存储的最大数值（255）。 但是如你所见，该字母能够显示到屏幕上，说明还是有一定的解决方法的。于是 编码 便出现了。

要用byte表示code point，我们需要在一定程度上对其进行编码。 Elixir使用UTF-8为默认编码格式。 当我们说某个字符串是UTF-8编码的二进制数据，意思是该字符串是一串byte， 以一定方法组织来表示特定的code points，即UTF-8编码。

因此当我们存储字母 `ł` 的时候，实际上是用两个bytes来表示它。 这就是为什么有时候对同一字符串，调用函数 `byte_size/1` 和 `String.length/1` 结果不一样：

```
iex> string = "hełło"
"hełło"
iex> byte_size string
7
iex> String.length string
5
```

UTF-8需要1个byte来表示code points：h，e和o，用2个bytes表示ł。 在Elixir中可以使用 `?` 运算符获取code point值：

```
iex> ?a
97
iex> ?ł
322
```

你还可以使用 String模块里的函数 将字符串切成单独的code points：

```
iex> String.codepoints("hełło")
["h", "e", "ł", "ł", "o"]
```

Elixir为字符串操作提供了强大的支持。实际上，Elixir通过了文章“字符串类型破了”记录的所有测试。

不仅如此，因为字符串是二进制，Elixir还提供了更强大的底层类型的操作。 下面就来介绍该底层类型---二进制。

6.2-二进制（和bitstring）

在Elixir中可以用 `<<>>` 定义一个二进制：

```
iex> <<0, 1, 2, 3>>
<<0, 1, 2, 3>>
iex> byte_size <<0, 1, 2, 3>>
4
```

一个二进制只是一连串bytes。这些bytes可以以任何方法组织，即使凑不成一个合法的字符串：

```
iex> String.valid?(<<239, 191, 191>>)
false
```

字符串的拼接操作实际上是二进制的拼接操作：

```
iex> <<0, 1>> <> <<2, 3>>
<<0, 1, 2, 3>>
```

一个常见技巧是，通过给某字符串尾部拼接一个null byte `<<0>>` ， 来看看该字符串内部二进制的样子：

```
iex> "hełło" <> <<0>>
<<104, 101, 197, 130, 197, 130, 111, 0>>
```

二进制中的每个数值都表示一个byte，因此其最大是255。 如果超出了255，二进制允许你再提供一个修改器（标识一下那个位置的存储空间大小）使其可以存储； 或者将其转换为utf8编码后的形式（变成多个byte的二进制）：

```
iex> <<255>>
<<255>>
iex> <<256>> # truncated
<<0>>
iex> <<256 :: size(16)>> # use 16 bits (2 bytes) to store the number
<<1, 0>>
iex> <<256 :: utf8>> # the number is a code point
"Ā"
iex> <<256 :: utf8, 0>>
<<196, 128, 0>>
```

如果一个byte是8 bits，那如果我们给一个size是1 bit的修改器会怎样？：

```
iex> <<1 :: size(1)>>
<<1::size(1)>>
iex> <<2 :: size(1)>> # truncated
<<0::size(1)>>
iex> is_binary(<< 1 :: size(1)>>)
false
iex> is_bitstring(<< 1 :: size(1)>>)
true
iex> bit_size(<< 1 :: size(1)>>)
1
```

这样（每个元素是1 bit）就不再是二进制（人家每个元素是byte，至少8 bits）了， 而是bitstring，就是一串比特！ 所以实际上二进制就是一串比特，只是比特数是8的倍数。

也可以对二进制或bitstring做模式匹配：

```
iex> <<0, 1, x>> = <<0, 1, 2>>
<<0, 1, 2>>
iex> x
2
iex> <<0, 1, x>> = <<0, 1, 2, 3>>
** (MatchError) no match of right hand side value: <<0, 1, 2, 3>>
```

注意（没有修改器标识的情况下）二进制中的每个元素都应该匹配8 bits。 因此上面最后的例子，匹配的左右两端不具有相同容量，因此出现错误。

下面是使用了修改器标识的匹配例子：

```
iex> <<0, 1, x :: binary>> = <<0, 1, 2, 3>>
<<0, 1, 2, 3>>
iex> x
<<2, 3>>
```

上面的模式仅在二进制 尾部 元素被修改器标识为又一个二进制时才正确。 字符串的连接操作也是一个意思：

```
iex> "he" <> rest = "hello"
"hello"
iex> rest
"llo"
```

总之，记住字符串是UTF-8编码的二进制，而二进制是特殊的、数量是8的倍数的bitstring。这种机制增加了Elixir在处理bits或bytes时的灵活性。 而现实中99%的时候你会用 `is_binary/1` 和 `byte_size/1` 函数跟二进制打交道。

6.3-字符列表

字符列表就是字符的列表。 双引号包裹字符串，单引号包裹字符列表。

```
iex> 'hełło'
[104, 101, 322, 322, 111]
iex> is_list 'hełło'
true
iex> 'hello'
'hello'
```

字符列表存储的不是bytes，而是字符的code points（实际上就是这些code points的普通列表）。 如果某字符不属于ASCII返回，iex就打印它的code point。

实际应用中，字符列表常被用来做参数，同一些老的库，或者同Erlang平台交互。 因为这些老库不接受二进制作为参数。 将字符列表和字符串之间转换，使用函数 `to_string/1` 和 `to_char_list/1` ：

```
iex> to_char_list "hełło"
[104, 101, 322, 322, 111]
iex> to_string 'hełło'
"hełło"
iex> to_string :hello
"hello"
iex> to_string 1
"1"
```

注意这些函数是多态的。它们不但转化字符列表和字符串，还能转化字符串和整数，等等。

7-键值列表-图-字典

到目前还没有讲到任何关联性数据结构，即那种可以将一个或几个值关联到一个key上。 不同语言有不同的叫法，如字典，哈希，关联数组，图，等等。

Elixir中有两种主要的关联性结构：键值列表（keyword list）和图（map）。

7.1-键值列表

在很多函数式语言中，常用二元元组的列表来表示关联性数据结构。在Elixir中也是这样。 当我们有了一个元组（不一定仅有两个元素的元组）的列表，并且每个元组的第一个元素是个原子， 那就称之为键值列表：

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
iex> list[:a]
1
```

> 当原子key和关联的值之间没有逗号分隔时，可以把原子的冒号拿到字母的后面。这时，原子与后面的数值之间要有一个空格。

如你所见，Elixir使用比较特殊的语法来定义这样的列表，但实际上它们会映射到一个元组列表。 因为它们是简单的列表而已，所有针对列表的操作，键值列表也可以用。

比如，可以用 `++` 运算符为列表添加元素：

```
iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

上面例子中重复出现了 `:a` 这个key，这是允许的。 以这个key取值时，取回来的是第一个找到的（因为有序）：

```
iex> new_list = [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex> new_list[:a]
0
```

键值列表十分重要，它有两大特点：

- 有序
- key可以重复（！仔细看上面两个示例）

例如，Ecto库使用这两个特点 写出了精巧的DSL（用来写数据库query）：

```
query = from w in Weather,
          where: w.prcp > 0,
          where: w.temp < 20,
        select: w
```

这些特性使得键值列表成了Elixir中为函数提供额外选项的默认手段。 在第5章我们讨论了 `if/2` 宏，提到了下方的语法：

```
iex> if false, do: :this, else: :that
:that
```

do: 和else: 就是键值列表！事实上代码等同于：

```
iex> if(false, [do: :this, else: :that])
:that
```

当键值列表是函数最后一个参数时，方括号就成了可选的。

为了操作关键字列表，Elixir提供了 键值（keyword）模块。 记住，键值列表就是简单的列表，和列表一样提供了线性的性能。 列表越长，获取长度或找到一个键值的速度越慢。 因此，关键字列表在Elixir中一般就作为函数调用的可选项。 如果你要存储大量数据，并且保证一个键只对应最多一个值，那就使用图。

对键值列表做模式匹配：

```
iex> [a: a] = [a: 1]
[a: 1]
iex> a
1
iex> [a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
iex> [b: b, a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
```

尽管如此，对列表使用模式匹配很少用到。因为不但要元素个数相等，顺序还要匹配。

7.2-图（maps）

无论何时想用键-值结构，图都应该是你的第一选择。Elixir中，用 `%{}` 定义图：

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
iex> map[2]
:b
```

和键值列表对比，图有两主要区别：

- 图允许任何类型值作为键
- 图的键没有顺序

如果你向图添加一个已有的键，将会覆盖之前的键-值对：

```
iex> %{1 => 1, 1 => 2}
%{1 => 2}
```

如果图中的键都是原子，那么你也可以用键值列表中的一些语法：

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
```

对比键值列表，图的模式匹配很是有用：

```
iex> %{} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> a
1
iex> %{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 => :b, :a => 1}
```

如上所示，图A与另一个图B做匹配。 图B中只要包含有图A的键，那么两个图就能匹配上。若图A是个空的，那么任意图B都能匹配上。 但是如果图B里不包含图A的键，那就匹配失败了。

图还有个有趣的功能：它提供了特殊的语法来修改和访问原子键：

```
iex> map = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> map.a
1
iex> %{map | :a => 2}
%{:a => 2, 2 => :b}
iex> %{map | :c => 3}
** (ArgumentError) argument error
```

使用上面两种语法要求的前提是所给的键是切实存在的。最后一条语句错误的原因就是键 `:c` 不存在。

未来几章中我们还将讨论结构体（structs）。结构体提供了编译时的保证，它是Elixir多态的基础。 结构体是基于图的，上面例子提到的修改键值的前提就变得十分重要。

图模块提供了许多关于图的操作。 它提供了与键值列表许多相似的API，因为这两个数据结构都实现了字典的行为。

> 图是最近连同EEP 43被引入Erlang虚拟机的。 Erlang 17提供了EEP的部分实现，只支持一小部分图功能。 这意味着图仅在存储不多的键时，图的性能还行。 为了解决这个问题，Elixir还提供了 HashDict模块。 该模块提供了一个字典来支持大量的键，并且性能不错。

7.3-字典（Dicts）

Elixir中，键值列表和图都被称作字典。 换句话说，一个字典就像一个接口（在Elixir中称之为行为behaviour）。 键值列表和图模块实现了该接口。

这个接口定义于Dict模块， 该模块还提供了底层实现的一个API：

```
iex> keyword = []
[]
iex> map = %{}
%{}
iex> Dict.put(keyword, :a, 1)
[a: 1]
iex> Dict.put(map, :a, 1)
%{a: 1}
```

字典模块允许开发者实现自己的字典形式，提供一些特殊的功能。 字典模块还提供了所有字典类型都可以使用的函数。 如， `Dicr.equal?/2` 可以比较两个字典类型（可以是不同的实现）。

你会疑惑些程序时用keyword，Map还是Dict模块呢？答案是：看情况。

如果你的代码期望接受一个关键字作为参数，那么使用简直列表模块。 如果你想操作一个图，那就使用图模块。 如果你想你的API对所有字典类型的实现都有用， 那就使用字典模块（确保以不同的实现作为参数测试一下）。

8-模块

Elixir中我们把许多函数组织成一个模块。我们在前几章已经提到了许多模块， 如String模块：

```
iex> String.length "hello"
5
```

创建自己的模块，用`defmodule`宏。用`def`宏在其中定义函数：

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

> 像ruby一样，模块名大写起头

8.1-编译

通常把模块写进文件，这样可以编译和重用。假如文件`math.ex`有如下内容：

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

这个文件可以用`elixirc`进行编译：

```
$ elixirc math.ex
```

这将生成名为`Elixir.Math.beam`的bytecode文件。 如果这时再启动iex，那么这个模块就已经可以用了（假如在含有该编译文件的目录启动iex）：

```
iex> Math.sum(1, 2)
3
```

Elixir工程通常组织在三个文件夹里：

- ebin，包括编译后的字节码
- lib，包括Elixir代码（.ex文件）
- test，测试代码（.exs文件）

实际项目中，构建工具Mix会负责编译，并且设置好正确的路径。 而为了学习方便，Elixir也提供了脚本模式，可以更灵活而不用编译。

8.2-脚本模式

除了.ex文件，Elixir还支持.exs脚本文件。 Elixir对两种文件一视同仁，唯一区别是.ex文件会保留编译执行后产出的比特码文件， 而.exs文件用来作脚本执行，不会留下比特码文件。例如，如下创建名为math.exs的文件：

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

执行之：

```
$ elixir math.exs
```

像这样执行脚本文件时，将在内存中编译和执行，打印出“3”作为结果。没有比特码文件生成。 后文中（为了学习和练习方便），推荐使用脚本模式执行学到的代码。

8.3-命名函数

在某模块中，我们可以用 `def/2` 宏定义函数，用 `defp/2` 定义私有函数。 用 `def/2` 定义的函数可以被其它模块中的代码使用，而私有函数仅在定义它的模块内使用。

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

Math.sum(1, 2)    #=> 3
Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

函数声明也支持使用卫兵或多个子句。 如果一个函数有好多子句，Elixir会匹配每一个子句直到找到一个匹配的。 下面例子检查参数是否是数字：

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_number(x) do
    false
  end
end

Math.zero?(0)  #=> true
Math.zero?(1)  #=> false

Math.zero?([1,2,3])
#=> ** (FunctionClauseError)
```

如果没有一个子句能匹配参数，会报错。

8.4-函数捕捉

本教程中提到函数，都是用 `name/arity` 的形式描述。 这种表示方法可以被用来获取一个命名函数（赋给一个函数型变量）。 下面用iex执行一下上文定义的math.exs文件：

```
$ iex math.exs
```

```
iex> Math.zero?(0)
true
iex> fun = &Math.zero?/1
&Math.zero?/1
iex> is_function fun
true
iex> fun.(0)
true
```

用 `&<function notation>` 通过函数名捕捉一个函数，它本身代表该函数值（函数类型的值）。 它可以不必赋给一个变量，直接用括号来使用该函数。

本地定义的，或者已导入的函数，比如 `is_function/1` ，可以不用前缀模模块名：

```
iex> &is_function/1
&:erlang.is_function/1
iex> (&is_function/1).(fun)
true
```

这种语法还可以作为快捷方式来创建和使用函数：

```
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2
```

代码中 `&1` 表示传给该函数的第一个参数。 因此，`&(&1+1)` 其实等同于 `fn x->x+1 end` 。在创建短小函数时，这个很方便。 想要了解更多关于 `&` 捕捉操作符，参考Kernel.SpecialForms文档。

8.5-默认参数

Elixir中，命名函数也支持默认参数：

```
defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

任何表达式都可以作为默认参数，但是只在函数调用时 用到了 才被执行。 （函数定义时，那些表达式只是存在那儿，不执行；函数调用时，没有用到默认值，也不执行）。

```
defmodule DefaultTest do
  def dowork(x \\ IO.puts "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
hello
:ok
```

如果有默认参数值的函数有了多条子句，推荐先定义一个函数头（无具体函数体）声明默认参数：

```
defmodule Concat do
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when is_nil(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello")               #=> Hello
```

使用默认值时，注意对函数重载会有一定影响。考虑下面例子：

```
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end

  def join(a, b, sep \\ " ") do
    IO.puts "***Second join"
    a <> sep <> b
  end
end
```

如果将以上代码保存在文件“concat.ex”中并编译，Elixir会报出以下警告：

```
concat.ex:7: this clause cannot match because a previous clause at line 2 always matches
```

编译器是在警告我们，在使用两个参数调用 `join` 函数时，总使用第一个函数定义。 只有使用三个参数调用时，才会使用第二个定义：

```
$ iex concat.exs
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_world"
```

后面几章将介绍使用命名函数来做循环，如何从别的模块中导入函数，以及模块的属性等。

9-递归

因为在Elixir中（或所有函数式语言中），数据有不变性（immutability），因此在写循环时与传统的命令式（imperative）语言有所不同。 例如某命令式语言的循环可以这么写：

```
for(i = 0; i < array.length; i++) {
  array[i] = array[i] * 2
}
```

上面例子中，我们改变了 `array` ，以及辅助变量 `i` 的值。这在Elixir中是不可能的。 尽管如此，函数式语言却依赖于某种形式的循环---递归：一个函数可以不断地被递归调用，直到某条件满足才停止。 考虑下面的例子：打印一个字符串若干次：

```
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

一个函数可以有许多子句（上面看起来定义了两个函数，但卫兵条件不同，可以看作同一个函数的两个子句）。 当参数匹配该子句的模式，且该子句的卫兵表达式返回true，才会执行该子句内容。 上面例子中，当 `print_multiple_times/2` 第一次调用时，n的值是3。

第一个子句有卫兵表达式要求n必须小于等于1。因为不满足此条件，代码找该函数的下一条子句。

参数匹配第二个子句，且该子句也没有卫兵表达式，因此得以执行。 首先打印 `msg` ，然后调用自身并传递第二个参数 `n-1` (=2)。 这样 `msg` 又被打印一次，之后调用自身并传递参数 `n-1` (=1)。

这个时候，n满足第一个函数子句条件。遂执行该子句，打印 `msg` ，然后就此结束。

我们称例子中第一个函数子句这种子句为“基本情形”。 基本情形总是最后被执行，因为起初通常都不匹配执行条件，程序而转去执行其它子句。 但是，每执行一次其它子句，条件都向这基本情形靠拢一点，直到最终回到基本情形处执行代码。

下面我们使用递归的威力来计算列表元素求和：

```
defmodule Math do
  def sum_list([head|tail], accumulator) do
    sum_list(tail, head + accumulator)
  end

  def sum_list([], accumulator) do
    accumulator
  end
end

Math.sum_list([1, 2, 3], 0) #=> 6
```

我们首先用列表[1,2,3]和初值0作为参数调用函数，程序将逐个匹配各子句的条件，执行第一个符合要求的子句。 于是，参数首先满足例子中定义的第一个子句。参数匹配使得head = 1，tail = [2,3]，accumulator = 0。

然后执行该字据内容，把head + accumulator作为第二个参数，连带去掉了head的列表做第一个参数，再次调用函数本身。 如此循环，每次都把新传入的列表的head加到accumulator上，传递去掉了head的列表。 最终传递的列表为空，符合第二个子句的条件，执行该子句，返回accumulator的值6。

几次函数调用情况如下：

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_list [], 6
```

这种使用列表做参数，每次削减一点列表的递归方式，称为“递减”算法，是函数式编程的核心。

如果我们想给每个列表元素加倍呢？：

```
defmodule Math do
  def double_each([head|tail]) do
    [head * 2| double_each(tail)]
  end

  def double_each([]) do
    []
  end
end

Math.double_each([1, 2, 3]) #=> [2, 4, 6]
```

此处使用了递归来遍历列表元素，使它们加倍，然后返回新的列表。 这样以列表为参数，递归处理其每个元素的方式，称为“映射（map）”算法。

递归和列尾调用优化（tail call optimization）是Elixir中重要的部分，通常用来创建循环。 尽管如此，在Elixir中你很少会使用以上方式来递归地处理列表。

下一章要介绍的Enum模块为操作列表提供了诸多方便。 比如，下面例子：

```
iex> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end)
6
iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
[2, 4, 6]
```

或者，使用捕捉的语法：

```
iex> Enum.reduce([1, 2, 3], 0, &+/2)
6
iex> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```

10-枚举类型和流

10.1-枚举类型

Elixir提供了枚举类型（enumerables）的概念，使用Enum模块操作它们。我们已经介绍过两种枚举类型：列表和图。

```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v end)
[2, 12]
```

Enum模块为枚举类型提供了大量函数来变化，排序，分组，过滤和读取元素。 Enum模块是开发者最常用的模块之一。

Elixir还提供了范围（range）：

```
iex> Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.reduce(1..3, 0, &+/2)
6
```

因为Enum模块在设计时为了适用于不同的数据类型，所以它的API被限制为多数据类型适用的函数。 为了实现某些操作，你可能需要针对某类型使用某特定的模块。 比如，如果你要在列表中某特定位置插入一个元素，要用List模块中的List.insert_at/3函数。而向某些类型内插入数据是没意义的，比如范围。

Enum中的函数是多态的，因为它们能处理不同的数据类型。 尤其是，模块中可以适用于不同数据类型的函数，它们是遵循了Enumerable协议。 我们在后面章节中将讨论这个协议。下面将介绍一种特殊的枚举类型：流。

10.2-积极vs懒惰

Enum模块中的所有函数都是积极的。多数函数接受一个枚举类型，并返回一个列表：

```
iex> odd? = &(rem(&1, 2) != 0)
#Function<6.80484245/1 in :erl_eval.expr/5>
iex> Enum.filter(1..3, odd?)
[1, 3]
```

这意味着当使用Enum进行多种操作时，每次操作都生成一个中间列表，直到得出最终结果：

```
iex> 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

上面例子是一个含有多个操作的管道。从一个范围开始，然后给每个元素乘以3。 该操作将会生成的中间结果是含有100000个元素的列表。 然后我们过滤掉所有偶数，产生又一个新中间结果：一个50000元素的列表。 最后求和，返回结果。

> 这个符号的用法似乎和F#中的不一样啊...

作为一个替代，流模块提供了懒惰的实现：

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

与之前Enum的处理不同，流先创建了一系列的计算操作。然后仅当我们把它传递给Enum模块，它才会被调用。流这种方式适用于处理大量的（甚至是无限的）数据集合。

10.3-流

流是懒惰的，比起Enum来说。 分步分析一下上面的例子，你会发现流与Enum的区别：

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [#Function<34.16982430/1 in Stream.map/2>]]>
```

流操作返回的不是结果列表，而是一个数据类型---流，一个表示要对范围1..100000使用map操作的动作。

另外，当我们用管道连接多个流操作时：

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
#Stream<[enum: 1..100000, funs: [...]]>
```

流模块中的函数接受任何枚举类型为参数，返回一个流。 流模块还提供了创建流（甚至是无限操作的流）的函数。 例如， `Stream.cycle/1` 可以用来创建一个流，它能无限周期性枚举所提供的参数（小心使用）：

```
iex> stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.cycle/1>
iex> Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

另一方面，`Stream.unfold/2`函数可以生成给定的有限值：

```
iex> stream = Stream.unfold("hełło", &String.next_codepoint/1)
#Function<39.75994740/2 in Stream.unfold/2>
iex> Enum.take(stream, 3)
["h", "e", "ł"]
```

另一个有趣的函数是`Stream.resource/3`，它可以用来包裹某资源，确保该资源在使用前打开，在用完后关闭（即使中途出现错误）。--类似C#中的use{}关键字。 比如，我们可以stream一个文件：

```
iex> stream = File.stream!("path/to/file")
#Function<18.16982430/2 in Stream.resource/3>
iex> Enum.take(stream, 10)
```

这个例子读取了文件的前10行内容。流在处理大文件，或者慢速资源（如网络）时非常有用。

一开始Enum和流模块中函数的数量多到让人气馁。但你会慢慢地熟悉它们。 建议先熟悉Enum模块，然后因为应用而转去流模块中那些相应的，懒惰版的函数。

11-进程

Elixir中，所有代码都在进程中执行。进程彼此独立，一个接一个并发执行，彼此通过消息传递来沟通。 进程不仅仅是Elixir中并发的基础，也是Elixir创建分布式、高容错程序的本质。

Elixir的进程和操作系统中的进程不可混为一谈。 Elixir的进程，在CPU和内存使用上，是极度轻量级的（不同于其它语言中的线程）。 因此，同时运行着数十万、百万个进程也并不是罕见的事。

本章将讲解如何派生新进程，以及在进程间如何发送和接受消息等基本知识。

11.1-进程派生

派生（spawning）一个新进程的方法是使用自动导入（kernel函数）的 `spawn/1` 函数：

```
iex> spawn fn -> 1 + 2 end
#PID<0.43.0>
```

函数 `spawn/1` 接收一个函数作为参数，在其派生出的进程中执行这个函数。

注意spawn/1返回一个PID（进程标识）。在这个时候，这个派生的进程很可能已经结束。 派生的进程执行完函数后便会结束：

```
iex> pid = spawn fn -> 1 + 2 end
#PID<0.44.0>
iex> Process.alive?(pid)
false
```

> 你可能会得到与例子中不一样的PID

用 `self/0` 函数获取当前进程的PID：

```
iex> self()
#PID<0.41.0>
iex> Process.alive?(self())
true
```

> 注：上文调用 `self/0` 加了括号。 但是如前文所说，在不引起误解的情况下，可以省略括号而只写 `self`

可以发送和接收消息，让进程变得越来越有趣。

11.2-发送和接收

使用 `send/2` 函数发送消息，用 `receive/1` 接收消息：

```
iex> send self(), {:hello, "world"}
{:hello, "world"}
iex> receive do
...>   {:hello, msg} -> msg
...>   {:world, msg} -> "won't match"
...> end
"world"
```

当有消息被发给某进程，该消息就被存储在该进程的邮箱里。 语句块 `receive/1` 检查当前进程的邮箱，寻找匹配给定模式的消息。 其中函数 `receive/1` 支持分支子句，如 `case/2` 。 当然也可以给子句加上卫兵表达式。

如果找不到匹配的消息，当前进程将一直等待，知道下一条信息到达。但是可以设置一个超时时间：

```
iex> receive do
...>   {:hello, msg} -> msg
...> after
...>   1_000 -> "nothing after 1s"
...> end
"nothing after 1s"
```

超时时间设为0表示你知道当前邮箱内肯定有邮件存在，很自信，因此设了这个极短的超时时间。

把以上概念综合起来，演示进程间发送消息：

```
iex> parent = self()
#PID<0.41.0>
iex> spawn fn -> send(parent, {:hello, self()}) end
#PID<0.48.0>
iex> receive do
...>   {:hello, pid} -> "Got hello from #{inspect pid}"
...> end
"Got hello from #PID<0.48.0>"
```

在shell中执行程序时，辅助函数 `flush/0` 很有用。它清空缓冲区，打印进程邮箱中的所有消息：

```
iex> send self(), :hello
:hello
iex> flush()
:hello
:ok
```

11.3-链接

Elixir中最常用的进程派生方式是通过函数 `spawn_link/1` 。在举例子讲解 `spawn_link/1` 之前，来看看如果一个进程失败了会发生什么：

```
iex> spawn fn -> raise "oops" end
#PID<0.58.0>
```

。。。啥也没发生。这时因为进程都是互不干扰的。如果我们希望一个进程中发生失败可以被另一个进程知道，我们需要链接它们。 使用 `spawn_link/1` 函数，例子：

```
iex> spawn_link fn -> raise "oops" end
#PID<0.60.0>
** (EXIT from #PID<0.41.0>) an exception was raised:
    ** (RuntimeError) oops
        :erlang.apply/2
```

当失败发生在shell中，shell会自动终止执行，并显示失败信息。这导致我们没法看清背后过程。 要弄明白链接的进程在失败时发生了什么，我们在一个脚本文件使用 `spawn_link/1` 并且执行和观察它：

```
# spawn.exs
spawn_link fn -> raise "oops" end

receive do
  :hello -> "let's wait until the process fails"
end
```

这次，该进程在失败时把它的父进程也弄停止了，因为它们是链接的。
手动链接进程： `Process.link/1` 。 建议可以多看看Process模块，里面包含很多常用的进程操作函数。

进程和链接在创建能高容错系统时扮演重要角色。在Elixir程序中，我们经常把进程链接到某“管理者”上。 由这个角色负责检测失败进程，并且创建新进程取代之。因为进程间独立，默认情况下不共享任何东西。 而且当一个进程失败了，也不会影响其它进程。 因此这种形式（进程链接到“管理者”角色）是唯一的实现方法。

其它语言通常需要我们来try-catch异常，而在Elixir中我们对此无所谓，放手任进程挂掉。 因为我们希望“管理者”会以更合适的方式重启系统。“要死你就快一点”是Elixir软件开发的通用哲学。

在讲下一章之前，让我们来看一个Elixir中常见的创建进程的情形。

11.4-状态

目前为止我们还没有怎么谈到状态。但是，只要你创建程序，就需要状态。 例如，保存程序的配置信息，或者分析一个文件先把它保存在内存里。 你怎么存储状态？

进程就是（最常见的）答案。我们可以写无限循环的进程，保存一个状态，然后通过收发信息来告知或改变该状态。 例如，写一个模块文件，用来创建一个提供k-v仓储服务的进程：

```
defmodule KV do
  def start do
    {:ok, spawn_link(fn -> loop(%{}) end)}
  end

  defp loop(map) do
    receive do
      {:get, key, caller} ->
        send caller, Map.get(map, key)
        loop(map)
      {:put, key, value} ->
        loop(Map.put(map, key, value))
    end
  end
end
```

注意 `start` 函数简单地派生一个新进程，这个进程以一个空的图为参数，执行 `loop/1` 函数。这个 `loop/1` 函数等待消息，并且针对每个消息执行合适的操作。 加入受到一个 `:get` 消息，它把消息发回给调用者，然后再次调用自身 `loop/1` ，等待新消息。 当受到 `:put` 消息，它便用一个新版本的图变量（里面的k-v更新了）再次调用自身。

执行一下试试：

```
iex> {:ok, pid} = KV.start
#PID<0.62.0>
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
nil
```

一开始进程内的图变量是没有键值的，所以发送一个 `:get` 消息并且刷新当前进程的收件箱，返回nil。 下面再试试发送一个 `:put` 消息：

```
iex> send pid, {:put, :hello, :world}
#PID<0.62.0>
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
```

注意进程是怎么保持一个状态的：我们通过同该进程收发消息来获取和更新这个状态。 事实上，任何进程只要知道该进程的PID，都能读取和修改状态。

还可以注册这个PID，给它一个名称。这使得人人都知道它的名字，并通过名字来向它发送消息：

```
iex> Process.register(pid, :kv)
true
iex> send :kv, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
```

使用进程维护状态，以及注册进程都是Elixir程序非常常用的方式。 但是大多数时间我们不会自己实现，而是使用Elixir提供的抽象实现。 例如，Elixir提供的agent就是一种维护状态的简单的抽象实现：

```
iex> {:ok, pid} = Agent.start_link(fn -> %{} end)
{:ok, #PID<0.72.0>}
iex> Agent.update(pid, fn map -> Map.put(map, :hello, :world) end)
:ok
iex> Agent.get(pid, fn map -> Map.get(map, :hello) end)
:world
```

给`Agent.start/2`方法加一个一个`:name`选项可以自动为其注册一个名字。 除了agents，Elixir还提供了创建通用服务器（generic servers，称作GenServer）、 通用时间管理器以及事件处理器（又称GenEvent）的API。 这些，连同“管理者”树，都可以在Mix和OTP手册里找到详细说明。

12-I/O

本章简单介绍Elixir的输入、输出机制以及相关的模块，如IO，文件 和路径。

现在介绍I/O似乎有点早，但是I/O系统可以让我们一窥Elixir哲学，满足我们对该语言以及VM的好奇心。

12.1-IO模块

IO模块是Elixir语言中读写标准输入、输出、标准错误、文件、设备的主要机制。使用该模块的方法颇为直接：

```
iex> IO.puts "hello world"
"hello world"
:ok
iex> IO.gets "yes or no? "
yes or no? yes
"yes\n"
```

IO模块中的函数默认使用标准输入输出。我们也可以传递 `:stderr` 来指示将错误信息写到标准错误设备上：

```
iex> IO.puts :stderr, "hello world"
"hello world"
:ok
```

12.2-文件模块

文件模块包含了可以让我们读写文件的函数。默认情况下文件是以二进制模式打开，它要求程序员使用特殊的 `IO.binread/2` 和 `IO.binwrite/2` 函数来读写文件：

```
iex> {:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
iex> IO.binwrite file, "world"
:ok
iex> File.close file
:ok
iex> File.read "hello"
{:ok, "world"}
```

文件可以使用 `:utf8` 编码打开，然后就可以被IO模块中其他函数使用了：

```
iex> {:ok, file} = File.open "another", [:write, :utf8]
{:ok, #PID<0.48.0>}
```

除了打开、读写文件的函数，文件模块还有许多函数来操作文件系统。 这些函数根据Unix功能相对应的命令命名。 如 `File.rm/1` 用来删除文件； `File.mkdir/1` 用来创建目录； `File.mkdir_p/1` 创建目录并保证其父目录一并创建； 还有 `File.cp_r/2` 和 `File.rm_rf/2` 用来递归地复制和删除整个目录。

你还会注意到文件模块中，一般函数都有一个名称类似的版本。区别是名称上一个有!(bang)一个没有。 例如，上面的例子我们在读取“hello”文件时，用的是不带!号的版本。 下面用例子演示下它们的区别：

```
iex> File.read "hello"
{:ok, "world"}
iex> File.read! "hello"
"world"
iex> File.read "unknown"
{:error, :enoent}
iex> File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
```

注意看，当文件不存在时，带!号的版本会报错。就是说不带!号的版本能照顾到模式匹配出来的不同情况。 但有的时候，你就是希望文件在那儿，!使得它能报出有意义的错误。

因此，不要写：

```
{:ok, body} = File.read(file)
```

相反地，应该这么写：

```
case File.read(file) do
  {:ok, body} -> # handle ok
  {:error, r} -> # handle error
end
```

或者

```
File.read!(file)
```

12.3-路径模块

文件模块中绝大多数函数都以路径作为参数。通常这些路径都是二进制，可以被路径模块提供的函数操作：

```
iex> Path.join("foo", "bar")
"foo/bar"
iex> Path.expand("~/hello")
"/Users/jose/hello"
```

有了以上介绍的几个模块和函数，我们已经能对文件系统进行基本的IO操作。 下面将讨论I/O模块中令人好奇的高级话题。这部分不是写Elixir程序必须掌握的，可以跳过不看。 但是如果你大概地浏览一下，可以了解IO是如何在VM上实现以及其它一些有趣的内容。

12.4-进程和组长

你可能已经发现，`File.open/2` 函数返回了一个包含PID的元祖：

```
iex> {:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
```

I/O模块实际上是同进程协同工作的。当你调用 `IO.write(pid, binary)` 时， I/O模块将发送一条消息给执行操作的进程。 让我们用自己的代码表述下这个过程：

```
iex> pid = spawn fn ->
...>  receive do: (msg -> IO.inspect msg)
...> end
#PID<0.57.0>
iex> IO.write(pid, "hello")
{:io_request, #PID<0.41.0>, #PID<0.57.0>, {:put_chars, :unicode, "hello"}}
** (ErlangError) erlang error: :terminated
```

调用 `IO.write/2` 之后，可以看见打印出了发给IO模块的请求。 然而因为我们没有提供某些东西，这个请求失败了。

StringIO模块 提供了一个基于字符串的IO实现：

```
iex> {:ok, pid} = StringIO.open("hello")
{:ok, #PID<0.43.0>}
iex> IO.read(pid, 2)
"he"
```

Erlang虚拟机用进程给I/O设备建模，允许同一个网络中的不同节点通过交换文件进程， 实现节点间的文件读写。在所有IO设备之中，有一个特殊的进程，称作组长（group leader）。

当你写东西到标准输出，实际上是发送了一条消息给组长，它把内容写给*STDIO*文件表述者：

```
iex> IO.puts :stdio, "hello"
hello
:ok
iex> IO.puts Process.group_leader, "hello"
hello
:ok
```

组长可为每个进程做相应配置，用于处理不同的情况。 例如，当在远程终端执行代码时，它能保证远程机器的消息可以被重定向到发起操作的终端上。

12.5-*iodata*和*chardata*

在以上所有例子中，我们都用的是二进制/字符串方式读写文件。 在“二进制、字符串和字符列表”那章里，我们注意到字符串就是普通的bytes， 而字符列表是code points（字符码）的列表。

I/O模块和文件模块中的函数接受列表作为参数。 还可以接受混合类型的列表，里面内容是整形、二进制都行：

```
iex> IO.puts 'hello world'
hello world
:ok
iex> IO.puts ['hello', ?\s, "world"]
hello world
:ok
```

尽管如此，有些地方还是要注意。一个列表可能表示一串byte，或者一串字符。 用哪一种需要看I/O设备是怎么编码的。

如果不指明编码，文件就以raw模式打开，这时候只能用文件模块里bin开头（二进制版） 的函数对其进行操作。 这些函数接受iodata*作为参数，即，它们期待一个整数值的列表，用来表示byte或二进制。

尽管只是细微的差别，如果你打算传递列表给那些函数，你需要考虑那些细节。 底层的bytes可以表示二进制，这种表示就是raw的。

13-别名和代码引用

为了实现软件重用，Elixir提供了三种指令（directives）。之所以称之为“指令”，是因为它们的作用域是词法作用域（lexicla scope）。

13.1-别名

宏 `alias` 可以为任何模块名设置别名。想象一下Math模块，它针对特殊的数学运算使用了特殊的列表实现：

```
defmodule Math do
  alias Math.List, as: List
end
```

现在，任何对 `List` 的引用将被自动变成对 `Math.List` 的引用。如果还想访问原来的 `List` ，需要前缀'Elixir'：

```
List.flatten             #=> uses Math.List.flatten
Elixir.List.flatten      #=> uses List.flatten
Elixir.Math.List.flatten #=> uses Math.List.flatten
```

> Elixir中定义的所有模块都在一个主Elixir命名空间。但是为方便起见，我们平时都不再前面加‘Elixir’。

别名常被使用于定义快捷方式。实际上不带 `as` 选项去调用 `alias` 会自动将这个别名设置为模块名的最后一部分：

```
alias Math.List
```

就相当于：

```
alias Math.List, as: List
```

注意，别名是词法作用域。即，你在某个函数中设置别名：

```
defmodule Math do
  def plus(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
  end
end
```

例子中 `alias` 指令只在函数 `plus/2` 中有用， `minus/2` 不受影响。

13.2-require

Elixir提供了许多宏，用于元编程（写能生成代码的代码）。

宏也是一堆代码，但它在编译时被展开和执行。 就是说为了使用一个宏，你需要确保它定义的模块和实现在编译期时可用。

要使用宏，需要用 `require` 指令引入定义宏的模块：

```
iex> Integer.odd?(3)
** (CompileError) iex:1: you must require Integer before invoking the macro Integer.odd?/
iex> require Integer
nil
iex> Integer.odd?(3)
true
```

Elixir中， `Integer.odd?/1` 函数被定义为一个宏，它可以被当作卫兵表达式（guards）使用。为了调用这个宏，首先得用 `require` 引用了*Integer*模块。

总的来说，宏在被用到之前不需要早早被require引用进来，除非我们想让这个宏在整个模块中可用。 尝试调用一个没有引入的宏会导致报错。 注意，像 `alias` 指令一样， `require` 也是词法作用域的。 下文中我们会进一步讨论宏。

13.3-import

当想轻松地访问别的模块中的函数和宏时，可以使用 `import` 指令加上宏定义模块的完整名字。 例如，如果我们想多次使用List模块中的 `duplicate` 函数，我们可以这么import它：

```
iex> import List, only: [duplicate: 2]
nil
iex> duplicate :ok, 3
[:ok, :ok, :ok]
```

这个例子中，我们只从List模块导入了函数 `duplicate/2` 。 尽管 `only:` 选项是可选的，但是推荐使用。

除了 `only:` 选项，还有 `except:` 选项。 使用选项 `only:` ，还可以传递给它 `:macros` 或 `:functions` 。 如下面例子，程序仅导入Integer模块中所有的宏：

```
import Integer, only: :macros
```

或者，仅导入所有的函数：

```
import Integer, only: :functions
```

注意， `import` 也是词法作用域，意味着我们可以在某特定函数中导入宏或方法：

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    # call duplicate
  end
end
```

在此例子中，导入的函数 `List.duplicate/2` 只在那个函数中可见。 该模块的其它函数中都不可用（自然，别的模块也不受影响）。

注意，若import一个模块，将自动require它。

13.4-别名机制

讲到这里你会问，Elixir的别名到底是什么，它是怎么实现的？

Elixir中的别名是以大写字母开头的标识符（像String, Keyword等等），在编译时会被转换为原子。 例如，别名‘String’会被转换为 `:"Elixir.String"` ：

```
iex> is_atom(String)
true
iex> to_string(String)
"Elixir.String"
iex> :"Elixir.String"
String
```

使用 `alias/2` 指令，其实只是简单地改变了这个别名将要转换的结果。

别名如此这般工作，是因为在Erlang虚拟机中（以及上面的Elixir），模块名是被表述成原子的。 例如，我们调用一个Erlang模块的机制是：

```
iex> :lists.flatten([1,[2],3])
[1, 2, 3]
```

这也是允许我们动态调用模块函数的机制：

```
iex> mod = :lists
:lists
iex> mod.flatten([1,[2],3])
[1,2,3]
```

一句话，我们只是简单地在原子 `:lists` 上调用了函数 `flatten` 。

13.5-嵌套

介绍了别名，现在可以讲讲嵌套（nesting）以及它在Elixir中是如何工作的。

考虑下面的例子：

```
defmodule Foo do
  defmodule Bar do
  end
end
```

该例子定义了两个模块*Foo*和*Foo.Bar*。 后者在*Foo*中可以用*Bar*为名来访问，因为它们在同一个词法作用域中。 如果之后开发者决定把Bar模块挪到另一个文件中，那它就需要以全名（Foo.Bar）或者别名来指代。

换句话说，上面的代码等同于：

```
defmodule Elixir.Foo do
  defmodule Elixir.Foo.Bar do
  end
  alias Elixir.Foo.Bar, as: Bar
end
```

13.6-多个

到Elixir v1.2，可以一次使用alias,import,require多个模块。这在建立Elixir程序的时候很常见，特别是使用嵌套的时候是最有用了。例如，假设你的程序所有模块都嵌套在 `MyApp` 下，需要使用别名 `MyApp.Foo` , `MyApp.Bar` 和 `MyApp.Baz` ，可以像下面一次完成：

```
alias MyApp.{Foo, Bar, Baz}
```

13.7- `use`

use用于请求在当前上下文中允许使用其他模块，是一个宏不是指令。开发者频繁使用 `use` 宏在当前上下文范围内引入其他功能，通常是模块。

例如，在编写测试时需要使用ExUni框架，开发者需要使用 `ExUnit.Case` 模块：

```
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    assert true
  end
end
```

在后面， `use` 请求需要的模块，然后调用 `__using__/1` 回调函数，允许模块在当前上下文中注入这些代码。总体说，如下模块：

```
defmodule Example do
  use Feature, option: :value
end
```

会编译成

```
defmodule Example do
  require Feature
  Feature.__using__(option: :value)
end
```

在以后章节我们可以看到，别名在宏机制中扮演了很重要的角色，来保证宏是干净的（hygienic）。

讨论到这里，模块基本上讲得差不多了。之后会讲解模块的属性。

14-模块属性

在Elixir中，模块属性（attributes）主要服务于三个目的：

1. 作为一个模块的注释，通常附加上用户或虚拟机用到的信息
2. 作为常量
3. 在编译时作为一个临时的存储机制

让我们一个一个讲解。

14.1-作为注释

Elixir从Erlang带来了模块属性的概念。例子：

```
defmodule MyServer do
  @vsn 2
end
```

这个例子中，我们显式地为该模块设置了 版本*(vsn即version)* 属性。 属性标识`@vsn`是预定义的属性名称，会被Erlang虚拟机的代码装载机制使用： 读取并检查该模块是否在某处被更新了。 如果不注明版本号，会被自动设置为这个模块函数的md5 checksum。

Elixir有个好多系统保留的预定义属性。比如一些常用的：

- @moduledoc 为整个模块提供文档说明
- @doc 为该属性后面的函数或宏提供文档说明
- @behaviour （注意这个单词是英式拼法）用来注明一个OTP或用户自定义行为
- @before_compile 提供一个每当模块被编译之前执行的钩子。这使得我们可以在模块被编译之前往里面注入函数。

@moduledoc和@doc是很常用的属性，推荐经常使用（写文档）。

Elixir视文档为一等公民，提供了很多方法来访问文档。

让我们回到上几章定义的Math模块，为它添加文档，然后依然保存在math.ex文件中：

```
defmodule Math do
  @moduledoc """
  Provides math-related functions.

  ## Examples

      iex> Math.sum(1, 2)
      3

  """

  @doc """
  Calculates the sum of two numbers.
  """
  def sum(a, b), do: a + b
end
```

上面例子使用了heredocs注释。heredocs是多行的文本，用三个引号包裹，保持里面内容的格式。 下面例子演示在iex中，用h命令读取模块的注释：

```
$ elixirc math.ex
$ iex
iex> h Math # Access the docs for the module Math
...
iex> h Math.sum # Access the docs for the sum function
...
```

Elixir还提供了ExDoc工具， 利用注释生成HTML页文档。

你可以看看模块 里面列出的模块属性列表，看看Elixir还支持那些模块属性。

Elixir还是用这些属性来定义 typespecs：

- @spec 为一个函数提供specification
- @callback 为行为回调函数提供spec
- @type 定义一个@spec中用到的类型
- @typep 定义一个私有类型，用于@spec
- @opaque 定义一个opaque类型用于@spec

本节讲了一些内置的属性。当然，属性可以被开发者、被一些类库扩展用来支持自定义的行为。

14.2-作为常量

Elixir开发者经常会将模块属性当作常量定义使用：

```
defmodule MyServer do
  @initial_state %{host: "147.0.0.1", port: 3456}
  IO.inspect @initial_state
end
```

> 不同于Erlang，默认情况下用户定义的属性不会被存储在模块里。属性值仅在编译时存在。 开发者可以通过调用`Module.register_attribute/3`来使属性的行为更接近Erlang。

访问一个未定义的属性会报警告：

```
defmodule MyServer do
  @unknown
end
warning: undefined module attribute @unknown, please remove access to @unknown or explici
```

最后，属性也可以在函数中被读取：

```
defmodule MyServer do
  @my_data 14
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end

MyServer.first_data #=> 14
MyServer.second_data #=> 13
```

注意，在函数内读取某属性，读取的是该属性当前值的快照。换句话说，读取的是编译时的值，而非运行时。 后面我们将看到，这个特点使得属性可以作为模块在编译时的临时存储。

14.3-作为临时存储

Elixir组织中有一个项目，叫做Plug。 这个项目的目标是创建一个通用的Web库和框架。

> 类似于ruby的rack

Plug库允许开发者定义它们自己的plug，可以在一个web服务器上运行：

```
defmodule MyPlug do
  use Plug.Builder

  plug :set_header
  plug :send_ok

  def set_header(conn, _opts) do
    put_resp_header(conn, "x-header", "set")
  end

  def send_ok(conn, _opts) do
    send(conn, 200, "ok")
  end
end

IO.puts "Running MyPlug with Cowboy on http://localhost:4000"
Plug.Adapters.Cowboy.http MyPlug, []
```

上面例子我们用了`plug/1`宏来连接各个在处理请求时会被调用的函数。 在内部，每当你调用`plug/1`时，Plug把参数存储在@plug属性里。 在模块被编译之前，Plug执行一个回调函数，这个函数定义了处理http请求的方法。 这个方法将顺序执行所有保存在@plug属性里的plugs。

为了理解底层的代码，我们需要宏。因此我们将回顾一下元编程手册里这种模式。 但是这里的重点是怎样使用属性来存储数据，让开发者得以创建DSL（领域特定语言）。

另一个例子来自ExUnit框架，它使用模块属性作为注释和存储：

```
defmodule MyTest do
  use ExUnit.Case

  @tag :external
  test "contacts external service" do
    # ...
  end
end
```

ExUnit中，@tag标签被用来注释该测试用例。之后，这些标签可以作为过滤测试用例之用。例如，你可以避免执行那些被标记成`:external`的测试，因为它们执行起来很慢。

本章带你一窥Elixir元编程的冰山一角，讲解了模块属性在开发中是如何扮演关键角色的。下一章将讲解结构体和协议。

15-结构体

在之前的几章中，我们谈到过图：

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
iex> map[:a]
1
iex> %{map | a: 3}
%{a: 3, b: 2}
```

结构体是基于图的一个扩展。它引入了默认值、编译期验证和多态性。

定义一个结构体，你只需在模块中调用 `defstruct/1` ：

```
iex> defmodule User do
...>   defstruct name: "john", age: 27
...> end
```

现在可以用 `%User()` 语法创建这个结构体的“实例”了：

```
iex> %User{}
%User{ name: "john", age: 27 }
iex> %User{ name: "meg" }
%User{ name: "meg", age: 27 }
iex> is_map(%User{})
true
```

结构体的编译期验证，指的是代码在编译时会检查结构体的字段存不存在：

```
iex> %User{ oops: :field }
** (CompileError) iex:3: unknown key :oops for struct User
```

当讨论图的时候，我们演示了如何访问和修改图现有的字段。结构体也是一样的：

```
iex> john = %User{}
%User{ name: "john", age: 27 }
iex> john.name
"john"
iex> meg = %{ john | name: "meg" }
%User{ name: "meg", age: 27 }
iex> %{ meg | oops: :field }
** (ArgumentError) argument error
```

使用这种修改的语法，虚拟机可以知道没有新的键增加到图/结构体中， 使得图可以在内存中共享它们的结构。在上面例子中，john和meg共享了相同的键结构。

结构体也能用在模式匹配中，它们保证结构体有相同的类型：

```
iex> %User{name: name} = john
%User{name: "john", age: 27}
iex> name
"john"
iex> %User{} = %{}
** (MatchError) no match of right hand side value: %{}
```

这里可以用模式匹配，是因为在结构体底层的图中有个叫 `__struct__` 的字段：

```
iex> john.__struct__
User
```

简单说，结构体就是个光秃秃的图外加一个默认字段。 但是，为图实现的协议都不能用于结构体。 例如，你不能枚举也不能用[]访问一个结构体：

```
iex> user = %User{}
%User{name: "john", age: 27}
iex> user[:name]
** (Protocol.UndefinedError) protocol Access not implemented for %User{age: 27, name: "jo
```

结构体也不是字典，因而也不能使用字典模块的函数：

```
iex> Dict.get(%User{}, :name)
** (ArgumentError) unsupported dict: %User{name: "john", age: 27}
```

下一章我们将介绍结构体是如何同协议进行交互的。

16-协议

协议是实现Elixir多态性的重要机制。任何数据类型只要实现了某协议，那么该协议的分发就是可用的。 让我们看个例子。

> 这里的“协议”二字对于熟悉ruby等具有duck-typing特性的语言的人来说会比较容易理解。

在Elixir中，只有false和nil被认为是false的。其它的值都被认为是true。 根据程序需要，有时需要一个 `blank?` 协议（注意，我们此处称之为“协议”）， 返回一个布尔值，以说明该参数是否为空。 举例来说，一个空列表或者空二进制可以被认为是空的。

我们可以如下定义协议：

```
defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"
  def blank?(data)
end
```

从上面代码的语法上看，这个协议 `Blank` 声明了一个函数 `blank?` ，接受一个参数。 看起来这个“协议”像是一份声明，需要后续的实现。 下面我们为不同的数据类型实现这个协议：

```
# 整型永远不为空
defimpl Blank, for: Integer do
  def blank?(_), do: false
end

# 只有空列表是“空”的
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_),  do: false
end

# 只有空map是“空”
defimpl Blank, for: Map do
  # 一定要记住，我们不能匹配 %{} ，因为它能match所有的map。
  # 但是我们能检查它的size是不是0
  # 检查size是很快速的操作
  def blank?(map), do: map_size(map) == 0
end

# 只有false和nil这两个原子被认为是空得
defimpl Blank, for: Atom do
  def blank?(false), do: true
  def blank?(nil),   do: true
  def blank?(_),     do: false
end
```

我们可以为所有内建数据类型实现协议：

- 原子
- BitString
- 浮点型
- 函数
- 整型
- 列表
- 图
- PID
- Port
- 引用
- 元祖

现在手边有了一个定义并被实现的协议，如此使用之：

```
iex> Blank.blank?(0)
false
iex> Blank.blank?([])
true
iex> Blank.blank?([1, 2, 3])
false
```

给它传递一个并没有实现该协议的数据类型，会导致报错：

```
iex> Blank.blank?("hello")
** (Protocol.UndefinedError) protocol Blank not implemented for "hello"
```

16.1-协议和结构体

协议和结构体一起使用能够加强Elixir的可扩展性。

在前面几章中我们知道，尽管结构体本质上就是图（map），但是它们和图并不共享各自协议的实现。 像前几章一样，我们先定义一个名为 `User` 的结构体：

```
iex> defmodule User do
...>   defstruct name: "john", age: 27
...> end
{:module, User, <<70, 79, 82, ...>>, {:__struct__, 0}}
```

然后看看能不能用刚才定义的协议：

```
iex> Blank.blank?(%{})
true
iex> Blank.blank?(%User{})
** (Protocol.UndefinedError) protocol Blank not implemented for %User{age: 27, name: "joh
```

果然，结构体没有使用协议针对图的实现。 因此，结构体需要使用它自己的协议实现：

```
defimpl Blank, for: User do
  def blank?(_), do: false
end
```

如果愿意，你可以定义你自己的语法来检查一个user是否为空。 不光如此，你还可以使用结构体创建更强健的数据类型（比如队列），然后实现所有相关的协议（就像枚举 `Enumerable` 那样），检查是否为空等等。

有些时候，程序员们希望给结构体提供某些默认的协议实现，因为显式给所有结构体都实现某些协议实在是太枯燥了。 这引出了下一节“回归一般化”（falling back to any）的说法。

16.2-回归一般化

能够给所有类型提供默认的协议实现肯定是很方便的。 在定义协议时，把 `@fallback_to_any` 设置为 `true` 即可：

```
defprotocol Blank do
  @fallback_to_any true
  def blank?(data)
end
```

现在这个协议可以被这么实现：

```
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

现在，那些我们还没有实现 `Blank` 协议的数据类型（包括结构体）也可以来判断是否为空了（虽然默认会被认为是false，哈哈）。

16.3-内建协议

Elixir自带了一些内建的协议。在前面几章中我们讨论过枚举模块，它提供了许多方法。 只要任何一种数据结构它实现了Enumerable协议，就能使用这些方法：

```
iex> Enum.map [1, 2, 3], fn(x) -> x * 2 end
[2,4,6]
iex> Enum.reduce 1..3, 0, fn(x, acc) -> x + acc end
6
```

另一个例子是 `String.Chars` 协议，它规定了如何将包含字符的数据结构转换为字符串类型。 它暴露为函数 `to_string` ：

```
iex> to_string :hello
"hello"
```

注意，在Elixir中，字符串插值操作里面调用了 `to_string` 函数：

```
iex> "age: #{25}"
"age: 25"
```

上面代码能工作，是因为25是数字类型，而数字类型实现了 `String.Chars` 协议。 如果传进去的是元组就会报错：

```
iex> tuple = {1, 2, 3}
{1, 2, 3}
iex> "tuple: #{tuple}"
** (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2, 3}
```

当想要打印一个比较复杂的数据结构时，可以使用 `inspect` 函数。该函数基于协议 `Inspect` ：

```
iex> "tuple: #{inspect tuple}"
"tuple: {1, 2, 3}"
```

Inspect 协议用来将任意数据类型转换为可读的文字表述。IEx用来打印表达式结果用的就是它：

```
iex> {1, 2, 3}
{1,2,3}
iex> %User{}
%User{name: "john", age: 27}
```

> `inspect` 是ruby中非常常用的方法。 这也能看出Elixir的作者们真是绞尽脑汁把Elixir的语法尽量往ruby上靠。

记住，头顶着#号被插的值，会被 `to_string` 表现成纯字符串。 在转换为可读的字符串时丢失了信息，因此别指望还能从该字符串取回原来的那个对象：

```
iex> inspect &(&1+2)
"#Function<6.71889879/1 in :erl_eval.expr/5>"
```

Elixir中还有些其它协议，但本章就讲这几个比较常用的。下一章将讲讲Elixir中的错误捕捉以及异常。

17-异常处理

Elixir有三种错误处理机制：errors，throws和exits。本章我们将逐个讲解它们，包括应该在何时使用哪一个。

17.1-Errors

举个例子，尝试让原子加上一个数字，就会激发一个错误（errors）：

```
iex> :foo + 1
** (ArithmeticError) bad argument in arithmetic expression
     :erlang.+(:foo, 1)
```

使用宏 `raise/1` 可以在任何时候激发一个运行时错误：

```
iex> raise "oops"
** (RuntimeError) oops
```

用 `raise/2` ，并且附上错误名称和一个键值列表可以激发规定好的错误：

```
iex> raise ArgumentError, message: "invalid argument foo"
** (ArgumentError) invalid argument foo
```

你可以使用 `defexception/2` 定义你自己的错误。最常见的是定义一个有消息说明的错误：

```
iex> defexception MyError, message: "default message"
iex> raise MyError
** (MyError) default message
iex> raise MyError, message: "custom message"
** (MyError) custom message
```

用 `try/catch` 结构可以处理异常：

```
iex> try do
...>   raise "oops"
...> rescue
...>   e in RuntimeError -> e
...> end
RuntimeError[message: "oops"]
```

这个例子处理了一个运行时异常，返回该错误本身（会被显示在IEx对话中）。 在实际操作中，Elixir程序员很少使用 `try/rescue` 结构。 例如，当文件打开失败，很多编程语言会强制你去处理一个异常。而Elixir提供的 `File.read/1` 函数返回包含信息的元组，不管文件打开成功与否：

```
iex> File.read "hello"
{:error, :enoent}
iex> File.write "hello", "world"
:ok
iex> File.read "hello"
{:ok, "world"}
```

这个例子中没有 `try/rescue` 。如果你想处理打开文件可能的不同结果，你可以使用case来匹配：

```
iex> case File.read "hello" do
...>   {:ok, body} -> IO.puts "got ok"
...>   {:error, body} -> IO.puts "got error"
...> end
```

使用这个匹配处理，你可以自己决定要不要把问题抛出来。 这就是为什么Elixir不让 `File.read/1` 等函数自己抛出异常。它把决定权留给程序员，让他们寻找最合适的处理方法。

如果你真的期待文件存在（打开文件时文件不存在这确实是一个错误），你可以简单地使用 `File.read!/1` ：

```
iex> File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
    (elixir) lib/file.ex:305: File.read!/1
```

换句话说，我们避免使用 `try/rescue` 是因为我们不用错误处理来控制程序执行流程。 在Elixir中，我们视错误为其字面意思：它们只不过是用来表示意外或异常的信息。
如果你真的希望改变执行过程，你可以使用 `throws` 。

17.2-Throws

在Elixir中，你可以抛出（throw）一个值稍后处理。 `throw` 和 `catch` 就被保留着为了处理一些你抛出了值，但是不用 `try/catch` 就取不到的情况。

这些情况实际中很少出现，除非当一个库的接口没有提供合适的API等情况。 例如，假如枚举模块没有提供任何API来寻找某范围内第一个13的倍数：

```
iex> try do
...>   Enum.each -50..50, fn(x) ->
...>     if rem(x, 13) == 0, do: throw(x)
...>   end
...>   "Got nothing"
...> catch
...>   x -> "Got #{x}"
...> end
"Got -39"
```

但是它提供了这样的函数`Enum.find/2`：

```
iex> Enum.find -50..50, &(rem(&1, 13) == 0)
-39
```

17.3-Exits

每段Elixir代码都在进程中运行，进程与进程相互交流。当一个进程终止了，它会发出`exit`信号。 一个进程可以通过显式地发出这个信号来终止：

```
iex> spawn_link fn -> exit(1) end
#PID<0.56.0>
** (EXIT from #PID<0.56.0>) 1
```

上面的例子中，链接着的进程通过发送`exit`信号（带有参数数字1）而终止。Elixir shell自动处理这个信息并把它们显示在终端上。

exit还可以被`try/catch`块捕获处理：

```
iex> try do
...>   exit "I am exiting"
...> catch
...>   :exit, _ -> "not really"
...> end
"not really"
```

因为`try/catch`已经很少用了，用它们捕获exit信号就更少见了。

exit信号是Erlang虚拟机提供的高容错性的重要部分。进程通常都在监督树（*supervision trees*）下运行。 监督树本身也是进程，它们通过exit信号监督其它进程。然后通过某些策略决定是否重启。

就是这种监督系统使得`try/catch`和`try/rescue`代码块很少用到。与其处理一个错误，不如让它快速失败。 因为在失败后，监督树会保证我们的程序将恢复到一个已知的初始状态去。

17.4-After

有时候有必要使用 `try/after` 来保证某资源在使用后被正确关闭或清除。 例如，我们打开一个文件，然后使用 `try/after` 来确保它在使用后被关闭：

```
iex> {:ok, file} = File.open "sample", [:utf8, :write]
iex> try do
...>   IO.write file, "olá"
...>   raise "oops, something went wrong"
...> after
...>   File.close(file)
...> end
** (RuntimeError) oops, something went wrong
```

17.5-变量作用域

对于定义在 `try/catch/rescue/after` 代码块中的变量，切记不可让它们泄露到外面去。这时因为 `try` 代码块有可能会失败，而这些变量此时并没有正常绑定数值：

```
iex> try do
...>   from_try = true
...> after
...>   from_after = true
...> end
iex> from_try
** (RuntimeError) undefined function: from_try/0
iex> from_after
** (RuntimeError) undefined function: from_after/0
```

至此我们结束了对 `try/catch/rescue` 等知识的介绍。你会发现其实这些概念在实际的Elixir编程中不太常用。尽管的确有时也会用到。

是时候讨论一些Elixir的概念，如列表速构（comprehensions）和魔法印（sigils）了。

18-列表速构（Comprehension）

> Comprehensions翻译成“速构”不知道贴不贴切，这参照了《Erlang/OTP in Action》译者的用辞。“速构”是函数式语言中常见的概念，它大体上指的是用一套规则（比如从另一个列表，过滤掉一些元素）来生成元素填充新列表。
> 这个概念我们在中学的数学课上就可能已经接触过，在大学高数中更为常见：如`{ x | x ∈ N }`，指的是这个集合里所有元素是自然数。 该集合是由自然数集合的元素映射生成过来的。相关知识可见WIKI。

Elixir中，使用枚举类型（如列表）来做循环操作是很常见的。对对象列表进行枚举时，通常要有选择性地过滤掉其中一些元素，还有可能做一些变换。列表速构（comprehensions）就是为此目的诞生的语法糖：把这些常见任务分组，放到特殊的`for`中执行。

例如，我们可以这样计算列表中每个元素的平方：

```
iex> for n <- [1, 2, 3, 4], do: n * n
[1, 4, 9, 16]
```

注意看，`<-`符号就是模拟自`∈`的形象。 这个例子用熟悉（当然，如果你高数课没怎么听那就另当别论）的数学符号表示就是：

```
S = { X^2 | X ∈ [1,4], X ∈ N}
```

这个例子用常见的编程语言去理解，基本上类似于foreach...in...什么的。但是更强大。

一个列表速构由三部分组成：生成器，过滤器和收集器。

18.2-生成器和过滤器

在上面的例子中，`n <- [1, 2, 3, 4]`就是生成器。它字面意思上生成了即将要在速构中使用的数值。 任何枚举类型都可以传递给生成器表达式的右端：

```
iex> for n <- 1..4, do: n * n
[1, 4, 9, 16]
```

这个例子中的生成器是一个范围。

生成器表达式支持模式匹配，它会忽略所有不匹配的模式。 想象一下如果不用范围而是用一个键值列表，键只有 `:good` 和 `:bad` 两种，来计算中间被标记成‘good’的元素的平方：

```
iex> values = [good: 1, good: 2, bad: 3, good: 4]
iex> for {:good, n} <- values, do: n * n
[1, 4, 16]
```

过滤器能过滤掉某些产生的值。例如我们可以只对奇数进行平方运算：

```
iex> require Integer
iex> for n <- 1..4, Integer.odd?(n), do: n * n
[1, 9]
```

过滤器会保留所有判断结果是非nil或非false的值。

总的来说，速构比直接使用枚举或流模块的函数提供了更精确的表述。 不但如此，速构还接受多个生成器和过滤器。下面就是一个例子，代码接受目录列表，删除这些目录下的所有文件：

```
for dir  <- dirs,
  file <- File.ls!(dir),
  path = Path.join(dir, file),
  File.regular?(path) do
    File.rm!(path)
end
```

需要记住的是，在速构中，变量赋值这种事应在生成器中进行。因为在过滤器或代码块中的赋值操作不会反映到速构外面去。

18.2-比特串生成器

速构也支持比特串作为生成器，而且这种生成器在组织处理比特串的流时非常有用。 下面的例子中，程序从二进制数据（表示为<<像素1的R值，像素1的G值，像素1的B值，像素2的R值，像素2的G...>>）中接收一个像素的列表，把它们转换为元组：

```
iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
iex> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

比特串生成器可以和“普通的”枚举类型生成器混合使用，过滤器也是。

18.3-Into

在上面的例子中，速构返回一个列表作为结果。 但是，通过使用 `:into` 选项，速构的结果可以插入到一个不同的数据结构中。 例如，你可以使用比特串生成器加上 `:into` 来轻松地构成无空格字符串：

```
iex> for <<c <- " hello world ">>, c != ?\s, into: "", do: <<c>>
"helloworld"
```

集合、图以及其他字典类型都可以传递给 `:into` 选项。总的来说， `:into` 接受任何实现了 *Collectable* 协议的数据结构。

例如，IO模块提供了流。流既是Enumerable也是Collectable。 你可以使用速构实现一个回声终端，让其返回任何输入的东西的大写形式：

```
iex> stream = IO.stream(:stdio, :line)
iex> for line <- stream, into: stream do
...>   String.upcase(line) <> "\n"
...> end
```

现在在终端中输入任意字符串，你会看到同样的内容以大写形式被打印出来。 不幸的是，这个例子会让你的shell陷入到该速构代码中，只能用Ctrl+C两次来退出。

19-魔法印(Sigils)

> 看看标题，这个“魔法印”是什么奇葩翻译？Sigil原意是“魔符，图章，印记”，如古代西方魔幻传说中的巫女、魔法师画的封印或者召唤魔鬼的六芒星，中国道士画的咒符，火影里面召唤守护兽的血印等。
> 在计算机编程领域，Sigil指的是在变量名称上做的标记，用来标明它的作用域或者类型什么的。例如某语言里面 `$var` 中的 `$` 就是这样的东西，表示其为全局变量。
> 这么看，翻译成“魔法印”还挺带感呢。

我们已经学习了Elixir提供的字符串（双引号包裹）和字符列表（单引号包裹）。但是对于Elixir中所有的文本描述型数据类型来说，这些只是冰山一角。其它的，例如原子也是一种文本描述型数据类型。

Elixir的一个特点就是高可扩展性：开发者能够为特定的领域来扩展语言。计算机科学的领域已是如此广阔。几乎无法设计一门语言来涵盖所有范围。我们的打算是，与其创造出一种万能的语言，不如创造一种可扩展的语言，让开发者可以根据所从事的领域来对语言进行扩展。

本章将讲述“魔法印（sigils）”，它是Elixir提供的处理文本描述型数据的一种机制。

19.1-正则表达式

魔法印以波浪号（~）起头，后面跟着一个字母，然后是分隔符。最常用的魔法印是~r，代表正则表达式：

```
# A regular expression that returns true if the text has foo or bar
iex> regex = ~r/foo|bar/
~r/foo|bar/
iex> "foo" =~ regex
true
iex> "bat" =~ regex
false
```

Elixir提供了Perl兼容的正则表达式（regex），由PCRE库实现。

正则表达式支持修饰符（modifiers），例如 `i` 修饰符使该正则表达式无视大小写：

```
iex> "HELLO" =~ ~r/hello/
false
iex> "HELLO" =~ ~r/hello/i
true
```

阅读Regex模块获取关于其它修饰符的及其所支持的操作的更多信息。

目前为止，所有的例子都用了 `/` 界定正则表达式。事实上魔法印支持8种不同的分隔符：

```
~r/hello/
~r|hello|
~r"hello"
~r'hello'
~r(hello)
~r[hello]
~r{hello}
~r<hello>
```

支持多种分隔符是因为在处理不同的魔法印的时候更加方便。 比如，使用括号作为正则表达式的分隔符会让人困惑，分不清括号是正则模式的一部分还是别的什么。 但是，括号对某些魔法印来说就很方便。

19.2-字符串、字符列表和单词魔法印

除了正则表达式，Elixir还提供了三种魔法印。

`~s` 魔法印用来生成字符串，类似双引号的作用：

```
iex> ~s(this is a string with "quotes")
"this is a string with \"quotes\""
```

通过这个例子可以看出，如果文本中有双引号，又不想逐个转义，可以用这种魔法印来包裹字符串。

`~c` 魔法印用来生成字符列表：

```
iex> ~c(this is a string with "quotes")
'this is a string with "quotes"'
```

`~w` 魔法印用来生成单词，以空格分隔开：

```
iex> ~w(foo bar bat)
["foo", "bar", "bat"]
```

`~w` 魔法印还接受 `c` ， `s` 和 `a` 修饰符（分别代表字符列表，字符串和原子）来选择结果的类型：

```
iex> ~w(foo bar bat)a
[:foo, :bar, :bat]
```

除了小写的魔法印，Elixir还支持大写的魔法印。如，`~s` 和 `~s` 都返回字符串，前者会解释转义字符而后者不会：

```
iex> ~s(String with escape codes \x26 interpolation)
"String with escape codes & interpolation"
iex> ~S(String without escape codes and without #{interpolation})
"String without escape codes and without \#{interpolation}"
```

字符串和字符列表支持以下转义字符：

- \" 表示一个双引号
- \' 表示一个单引号
- \\ 表示一个反斜杠
- \a 响铃
- \b 退格
- \d 删除
- \e 退出
- \f 换页
- \n 新行
- \r 换行
- \s 空格
- \t 水平制表符
- \v 垂直制表符
- \DDD, \DD, \D 八进制数字（如\377）
- \xDD 十六进制数字（如\x13）
- \x{D...} 多个十六进制字符的十六进制数（如\x{abc13}

魔法印还支持多行文本（heredocs），使用的是三个双引号或单引号：

```
iex> ~s"""
...> this is
...> a heredoc string
...> """
```

最常见的有多行文本的魔法印就是写注释文档了。 例如，如果你要在注释里写一些转义字符，这有可能会报错。

```
@doc """
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\\\"foo\\\"")
    "'foo'"

"""
def convert(...)
```

使用 `~S` ，我们就可以避免问题：

```
@doc ~S"""
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\"foo\"")
    "'foo'"

"""
def convert(...)
```

19.3-自定义魔法印

本章开头提到过，魔法印是可扩展的。事实上，魔法印 `~r/foo/i` 等于是用两个参数调用了函数 `sigil_r` ：

```
iex> sigil_r(<<"foo">>, 'i')
~r"foo"i
```

就是说，我们可以通过该函数阅读魔法印 `~r` 的文档：

```
iex> h sigil_r
...
```

我们也可以通过实现相应的函数来提供我们自己的魔法印。例如，我们来实现一个 `~i(N)` 魔法印，返回整数：

```
iex> defmodule MySigils do
...>   def sigil_i(string, []), do: String.to_integer(string)
...> end
iex> import MySigils
iex> ~i("13")
13
```

魔法印通过宏，可以用来做一些发生在编译时的工作。例如，正则表达式在编译时会被编译，而在执行的时候就不必再被编译了。 如果你对此主题感兴趣，可以多阅读关于宏的资料，并且阅读Kernel模块中那些魔法印的实现。

20-下一步

构建你第一个Elixir工程
社区和其它资源
Erlang一瞥

还想学习更多？继续阅读吧！

20.1-构建你第一个Elixir工程

Elixir提供了一个构建工具叫做Mix。你可以简单地执行以下命令来开始你的项目：

```
mix new path/to/new/project
```

我们已经写好了一份手册，涵盖了如何构建一个Elixir程序，包括它自己的监督树，配置，测试等等。 这个程序是一个分布式的键值对存储程序：

- Mix and OTP

20.2-社区和其它资源

Blog
文档

Seriously, please check out my next works about advanced Elixir programming !

请看拙作《高级Elixir编程》.名字随便起的，主要是Elixir编程一些进阶的知识。翻译、整理自官网或者自己理解等。 https://github.com/straightdave/elixir_adv

elixir进阶（Mix和OTP入门）

> 作者：straightdave
>
> 来源：advanced_elixir

Elixir是基于Erlang虚拟机的语言。 要真正发挥Elixir高可用、高并发等等优势，需要了解和学习E在它背后Erlang世界的知识。 本repo的文章，也属于一个入门。

- 1-Mix简介
- 2-Agent
- 3-GenServer
- 4-GenEvent
- 5-监督者和应用程序
- 6-ETS
- 7-依赖和伞工程
- 8-任务和gen_tcp
- 9-文档，测试和管道
- 10-分布式任务和配置

1-Mix简介

在这份指导手册中，我们将学习创建一个完整的Elixir应用程序，以及监督树、配置、测试等高级概念。

这个程序是一个分布式的键-值存储数据库。我们会把键-值存储在“桶”中，分布存储到多个节点。 我们还会创建一个简单的客户端工具，可以连接任意一个节点，并且能够发送类似以下的命令：

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

为了编写这个程序，我们将主要用到以下三个工具：

- **OTP(Open Telecom Platform)** OTP是一个随Erlang发布的代码库集合。Erlang开发者使用OTP来创建健壮的、高度容错的程序。在本章中，我们将来探索与Elixir整合在一起的OTP，包括监督树、事件管理等等；
- **Mix** Mix是随Elixir发布的构建工具，用来创建、编译、测试你的应用程序，还可以用来管理依赖等；
- **ExUnit** ExUnit是一个随Elixir发布的单元测试工具

本章会用Mix来创建我们第一个工程，探索OTP、Mix以及ExUnit的各种特性。

> 注意：
> 本手册需要Elixir v0.15.0（1.2.0发布后，这里被改为1.2.0了）或以上。 你可以使用命令`elixir -v`查看版本。 如果需要，可以参考《Elixir入门手册》第一章节内容安装最新的版本。
> 如果发现任何错误，请开issue或者发pull request。

1.1-第一个应用程序

当你安装Elixir时，你不仅得到了 `elixir` ， `elixirc` 和 `iex` 命令， 还得到一个可执行的Elixir脚本叫做 `mix` 。

从命令行输入 `mix new` 命令来创建我们的第一个工程。 我们需要传递工程名称作为参数（在这里，比如叫做 `kv` ）， 然后告诉mix我们的主模块的名字是全大写的 `KV` 。 否则按照默认，mix会创建一个主模块，名字是第一个字母大写的工程名称（ `Kv` ）。 因为K和V的含义在我们的程序中上是平等关系，所以最好是都用大写：

```
$ mix new kv --module KV
```

Mix将创建一个文件夹名叫 `kv` ，里面有一些文件：

```
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/kv.ex
* creating test
* creating test/test_helper.exs
* creating test/kv_test.exs
```

现在简单看看这些创建的文件。

> 注意：
> Mix是一个Elixir可执行脚本。这意味着，你要想用mix为名直接调用它， 需要提前将Elixir目录放进系统的环境变量中。否则，你需要使用elixir来执行mix：
>
> ```
> $ bin/elixir bin/mix new kv --module KV
> ```
>
> 你也可以用 `-S` 选项来执行elixir，它不管你有没有把mix的目录加入环境变量：
>
> ```
> $ bin/elixir -S mix new kv --module KV
> ```

1.2-工程的编译

一个名叫 `mix.exs` 的文件会被自动创建在工程目录中。 它的主要作用是配置你的工程。它的内容如下（略去代码中的注释）：

```
defmodule KV.Mixfile do
  use Mix.Project

  def project do
    [app: :kv,
     version: "0.0.1",
     elixir: "~> 1.2",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

我们的 `mix.exs` 定义了两个公共函数： 一个是 `project` ，它返回工程的配置信息，如工程名称和版本； 另一个是 `application` ，它用来生成应用程序文件。

还有一个私有函数叫做 `deps` ，它被 `project` 函数调用，里面定义了工程的依赖。 不一定非要把 `deps` 定义为一个独立的函数，但是这样做可以使工程的配置文件看起来整洁美观。

Mix还生成了文件 `lib/kv.ex` ，其内容是个简单的模块定义：

```
defmodule KV do
end
```

以上这个结构就足以编译我们的工程了：

```
$ cd kv
$ mix compile
```

将生成：

```
Compiled lib/kv.ex
Generated kv app
Consolidated List.Chars
Consolidated Collectable
Consolidated String.Chars
Consolidated Enumerable
Consolidated IEx.Info
Consolidated Inspect
```

注意文件 `lib/kv.ex` 被编译，生成了程序manifest文件： `kv.app` 及一些 协议(参考入门手册)。根据 `mix.exs` 的配置，所有编译产出被放在 `_build` 目录中。

一旦工程被编译成功，便可以从工程目录启动一个 `iex` 会话：

```
$ iex -S mix
```

1.3-执行测试

Mix还生成了合适的文件结构，来测试我们的工程。Mix工程一般沿用一些命名规则：在 `test` 目录中，测试文件一般以 `<filename>_test.exs` 模式命名。 每一个 `<filename>` 对应一个 `lib` 目录中的文件名。 根据这个命名规则，我们已经有了测试 `lib/kv.ex` 所需的 `test/kv_test.exs` 文件。 只是目前它几乎什么也没做：

```
defmodule KVTest do
  use ExUnit.Case
  doctest KV

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

需要注意几点：

1. 测试文件使用的扩展名(.exs)即Elixir脚本文件。这很方便，我们不用在跑测试前还编译一次。
2. 我们定义了一个测试模块名为 `KVTest` ，用 `ExUnit.Case` 来注入测试API， 并使用宏 `test/2` 定义了一个简单的测试；

Mix还生成了一个文件名叫 `test/test_helper.exs` ，它负责设置测试框架：

```
ExUnit.start()
```

每次Mix执行测试时，这个文件将自动被导入（required）。执行测试，使用命令 `mix test` ：

```
Compiled lib/kv.ex
Generated kv app
[...]
.

Finished in 0.04 seconds (0.04s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 540224
```

注意，每次运行 `mix test` 时，Mix会重新编译源文件，生成新的应用程序。 这是因为Mix支持多套执行环境，我们稍后章节会详细介绍。

另外，ExUnit为每一个成功的测试结果打印一个点，它还会自动随机安排测试顺序。 让我们把测试改成失败看看会发生啥。修改 `test/kv_test.exs` 里面的断言，改成：

```
assert 1 + 1 == 3
```

现在再次运行 `mix test` （注意这次没有编译行为发生）：

```
  1) test the truth (KVTest)
     test/kv_test.exs:5
     Assertion with == failed
     code: 1 + 1 == 3
     lhs:  2
     rhs:  3
     stacktrace:
       test/kv_test.exs:6

Finished in 0.05 seconds (0.05s on load, 0.00s on tests)
1 tests, 1 failures
```

ExUnit会为每个失败的测试结果打印一个详细的报告。其内容包含了测试名称，失败的代码， 失败断言中 `==` 号的左值（lhs）和右值(rhs)。

在错误提示的第二行（测试名称下面那行），是该测试的代码位置。 将这个位置作为参数给 `mix test` 命令，则将仅执行该条测试：

```
$ mix test test/kv_test.exs:5
```

这个十分有用是吧。

最后是关于错误的追踪栈信息，给出关于测试的额外信息。 包括测试失败的地方，还有原文件中产生失败的具体位置等。

1.4-环境

Mix支持“环境”的概念。它允许开发者为某些场景定义不同的编译等动作。 默认地，Mix理解三种环境：

- `:dev` - Mix任务的默认执行环境（如编译等操作）
- `:test` - `mix test` 使用的环境
- `:prod` - 用来将应用程序发布到产品环境

环境配置只对当前工程有效。我们之后会看到，向工程中添加的依赖默认在 `:prod` 环境下工作。

可以通过访问 `mix.exs` 工程配置文件中的 `Mix.env` 函数定义不同的环境配置， 它会以原子形式返回当前的环境。 比如我们用之于 `:build_embedded` 和 `:start_permanent:` 这两个选项：

```
def project do
  [...,
   build_embedded: Mix.env == :prod,
   start_permanent: Mix.env == :prod,
   ...]
end
```

上面代码的含义就是程序在 `:prod` 环境中运行的话，则使用那两个选项。

当你编译代码的时候，Elixir把编译产出都置于 `_build` 目录。 但是，有些时候Elixir为了避免一些不必要的复制操作， 会在 `_build` 目录中创建一些链接指向特定文件而不是copy。当 `:build_embedded` 选项被设置为true时可以制止这种行为， 从而在 `_build` 目录中提供执行程序所需的所有文件。

类似地，当 `:start_permanent` 选项设置为true的时候，程序会以“Permanent模式”执行。 意思是如果你的程序的监督树挂掉，Erlang虚拟机也会挂掉。 注意在:dev和:test环境中，我们可能不需要这样的行为。 因为在这些环境中，为了troubleshooting等目的，需要保持虚拟机持续运行。

Mix默认使用 `:dev` 环境，除非在执行测试时需要用到 `:test` 环境。 环境可以随时更改：

```
$ MIX_ENV=prod mix compile
```

或在Windows上：

```
> set /a "MIX_ENV=prod" && mix compile
```

1.5-探索

关于Mix，内容还有很多，我们在编写这个工程的过程中还会陆续接触到一些。 详细信息可以参考Mix的文档。

记住，你可以使用mix的帮助信息来帮助理解一些任务的操作方法，如：

```
$ mix help TASK
```

2-Agent

本章我们将创建一个名为 `KV.Bucket` 的模块。这个模块负责存储可被不同进程读写的键值对。

如果你跳过了“入门”手册，或者是太久以前读的，那么建议你最好重新阅读一下关于 进程 的那一章。 它是本节所内容的起点。

2.1-状态的麻烦

Elixir是一种“（变量值）不可变”的语言。默认情况下，没有什么是被共享的。 如果想要提供某种状态，通过其创建可以从不同地方访问的“桶”，我们有两种选择：

- 进程
- ETS（Erlang Term Storage）

我们之前介绍过进程，但ETS是个新东西，在后面的章节中再去探讨。 而当用到进程时，我们很少会去自己动手从底层做起，而是用Elixir和OTP中抽象出来的东西代替：

- Agent - 对状态简单的封装
- GenServer - “通用的服务器”（进程） 。它封装了状态，提供了同步或异步调用，支持代码热更新等等
- GenEvent - “通用的事件”管理器，允许向多个接收者发布事件消息
- Task - 计算处理的异步单元，可以派生出进程并稍后收集计算结果

我们在本“进阶”手册中会逐一讨论这些抽象物。 记住它们都是在进程基础上实现的，使用Erlang虚拟机提供的基本特性， 如 `send` ， `receive` ， `spawn` 和 `link` 。

2.2-Agents

Agent是对状态简单的封装。 如果你想要一个可以保存状态的地方（进程） ，那么Agent就是不二之选。 让我们在工程里启动一个 `iex` 对话：

```
$iex -S mix
```

然后“玩弄”一下Agent：

```
iex> {:ok, agent} = Agent.start_link fn -> [] end
{:ok, #PID<0.57.0>}
iex> Agent.update(agent, fn list -> ["eggs"|list] end)
:ok
iex> Agent.get(agent, fn list -> list end)
["eggs"]
iex> Agent.stop(agent)
:ok
```

这里用某个初始状态（空列表）启动了一个agent，然后执行了一个命令来修改这个状态， 加了一个新的列表项到头部。 `Agent.update/3` 的第二个参数是一个匿名函数： 它使用agent当前状态为输入，返回想要的新状态。 最终我们获取整个列表。 `Agent.get/3` 函数的第二个参数是个匿名函数： 它使用当前状态为输入，返回的值就是 `Agent.get/3` 的返回值。 一旦我们用完agent，我们调用 `Agent.stop/1` 来终止agent进程。

现在我们用Agent来实现 `KV.Bucket` 。当时在开始之前，我们先写些测试。 新建文件 `test/kv/bucket_test.exs` （回想一下 `.exs` 文件），内容是：

```
defmodule KV.BucketTest do
  use ExUnit.Case, async: true

  test "stores values by key" do
    {:ok, bucket} = KV.Bucket.start_link
    assert KV.Bucket.get(bucket, "milk") == nil

    KV.Bucket.put(bucket, "milk", 3)
    assert KV.Bucket.get(bucket, "milk") == 3
  end
end
```

我们的第一条测试很直白：启动一个 `KV.Bucket` ，然后执行 `get/2` 和 `put/2` 操作。 最后判断结果。我们不需要显式地停止agent进程。 因为该test里面用到的agent进程是链接到测试进程的，测试进程一结束它就会跟着结束。

同时还要注意我们向 `ExUnit.Case` 传递了一个 `async:true` 的选项。 这个选项使得该测试用例与其它同样包含 `:async` 选项的测试用例并行执行。 这种方式能够更好地利用计算机多核的能力。但要注意，这样的话，测试用例不能依赖或改变某些全局的值。 比如测试需要向文件系统里写入文字，或者注册进程，或者访问数据库等。 你在放置 `:async` 标记前必须考虑会不会在两个测试之间造成资源竞争。

不管是不是异步执行的，很明显我们的测试会失败，因为该实现的功能一个都没实现。

为了修复失败的用例，我们来创建文件 `lib/kv/bucket.ex` ，输入以下内容。 你可以不看下方的代码，自己随便尝试着创建agent的行为：

```
defmodule KV.Bucket do
  @doc """
  Starts a new bucket.
  """
  def start_link do
    Agent.start_link(fn -> %{} end)
  end

  @doc """
  Gets a value from the `bucket` by `key`.
  """
  def get(bucket, key) do
    Agent.get(bucket, &Map.get(&1, key))
  end

  @doc """
  Puts the `value` for the given `key` in the `bucket`.
  """
  def put(bucket, key, value) do
    Agent.update(bucket, &Map.put(&1, key, value))
  end
end
```

我们使用图(Map)来存储我们的键和值。函数捕捉符号 `&` 在《入门》中介绍过。 现在 `KV.Bucket` 模块定义好了，测试都通过了！你可以执行 `mix test` 试试。

2.3-ExUnit回调函数

在继续为 `KV.Bucket` 加入更多功能之前，先讲一讲ExUnit的回调函数。 你可能已经想到，每一个 `KV.Bucket` 的测试用例都需要用到bucket。 它要在该测试用例启动时设置好，还要在该测试用例结束时停止。 幸运的是，ExUnit支持回调函数，使我们跳过这重复机械的任务。

让我们使用回调机制重写刚才的测试：

```
defmodule KV.BucketTest do
  use ExUnit.Case, async: true

  setup do
    {:ok, bucket} = KV.Bucket.start_link
    {:ok, bucket: bucket}
  end

  test "stores values by key", %{bucket: bucket} do
    assert KV.Bucket.get(bucket, "milk") == nil

    KV.Bucket.put(bucket, "milk", 3)
    assert KV.Bucket.get(bucket, "milk") == 3
  end
end
```

我们首先利用 `setup/1` 宏，创建了设置bucket的回调函数。 这个函数会在每条测试用例执行前被执行一次，并且是与测试在同一个进程里。

注意我们需要一个机制来传递创建好的 `bucket` 的pid给测试用例。 我们使用 测试上下文 来达到这个目的。 当在回调函数里返回 `{:ok, bucket: bucket}` 的时候， ExUnit会把该返回值元祖（字典）的第二个元素merge进测试上下文中。 测试上下文是一个图，我们可以在测试用例

的定义中匹配它，从而获取这个上下文的值给用例中的代码使用：

```
test "stores values by key", %{bucket: bucket} do
  # `bucket` is now the bucket from the setup block
end
```

更多信息可以参考ExUnit.Case模块文档， 以及回调函数。

2.4-其它Agent行为

除了“读取”或者“修改”agent的状态，agent还允许我们使用 函数 `Agent.get_and_update/2` “读取并修改”它维持的状态。 我们用这个函数来实现删除 `KV.Bucket.delete/2` 功能---从bucket中删除一个值，并返回该值：

```
@doc """
Deletes `key` from `bucket`.

Returns the current value of `key`, if `key` exists.
"""
def delete(bucket, key) do
  Agent.get_and_update(bucket, &Map.pop(&1, key))
end
```

现在轮到你来给上面的代码写个测试啦。 你可以阅读Agent模块的文档 获取更多信息。

2.5-Agent中的C/S模式

在进入下一章之前，让我们讨论一下agent中的C/S二元模式。 先来展开刚刚写好的 `delete/2` 函数：

```
def delete(bucket, key) do
  Agent.get_and_update(bucket, fn dict->
    Map.pop(dict, key)
  end)
end
```

我们传递给agent的函数中的任何东西，都会出现在agent的进程里。 在这里，因为agent进程负责接收和回复我们的消息，因此可以说agent进程就是个服务器（服务端）。 而那个方法之外的任何东西，都被看成是在客户端的范围内。

这个区别很重要。如果有大量的工作要做，你必须考虑这个工作是放在客户端还是在服务器上执行。比如：

```
def delete(bucket, key) do
  :timer.sleep(1000) # puts client to sleep
  Agent.get_and_update(bucket, fn dict ->
    :timer.sleep(1000) # puts server to sleep
    Map.pop(dict, key)
  end)
end
```

当服务器上执行一个很耗时的工作时，所有其它对该服务器的请求都必须等待，直到那个工作完成。 这会造成客户端的超时。

下一章我们会探索通用服务器GenServer，它在概念上对服务器与客户端的隔离更明显。

3-通用服务器（GenServer）

上一章我们用agent实现了buckets。根据第一章所描述的，我们的设计是要给每个bucket赋予名字， 从而可以这么去访问：

```
CREATE shopping
OK

PUT shopping milk 1
OK

GET shopping milk
1
OK
```

因为agent是进程，每个bucket只有一个进程id（pid）而不是名字。 不过在《入门手册》中的进程那章中提到过，我们可以给进程注册名字。 我们貌似可以使用这个方法来给bucket起名：

```
iex> Agent.start_link(fn -> [] end, name: :shopping)
{:ok, #PID<0.43.0>}
iex> KV.Bucket.put(:shopping, "milk", 1)
:ok
iex> KV.Bucket.get(:shopping, "milk")
1
```

但是这是个很差的主意！在Elixir中，进程的名字存储为原子。 这意味着我们从外部客户端输入的bucket名字，都会被转换成原子。 记住，绝对不要把用户输入转换为原子。这是因为原子不会被垃圾收集器收集。 一旦原子被创建，它就不会被取消（你也没法主动释放一个原子，对吧）。 使用用户输入生成原子就意味着用户可以插入足够不同的名字来耗尽系统内存空间！

在实际操作中，在它用完内存之前会先触及Erland虚拟机的最大原子数量，从而造成系统崩溃。

比起滥用名字注册机制，我们可以创建我们自己的 注册表进程*(registry process)* 来维护一个字典， 用该字典联系起每个bucket的名字和进程。

这个注册表要能够保证永远处于最新状态。如果有一个bucket进程因故崩溃，注册表必须清除该进程信息， 以防止继续服务下次查找请求。 在Elixir中，我们描述这种情况会说“该注册表需要监视（monitor）每个bucket”。

我们将使用GenServer 来创建一个可以监视bucket进程的注册表进程。

在Elixir和OTP中，GenServer是创建“通用的服务器（generic servers）”的首选抽象物。

3.1-第一个GenServer

一个GenServer实现分为两个部分：客户端API和服务端回调函数。 这两部分可以写在同一个模块里，也可以分开写到两个模块中。 客户端和服务端运行于不同进程，依靠调用客户端函数来与服务端来回传递消息。 方便起见，这里我们将这两部分写在一个模块中。 创建文件 `lib/kv/registry.ex` ，包含以下内容：

```
defmodule KV.Registry do
  use GenServer

  ## Client API

  @doc """
  Starts the registry.
  """
  def start_link() do
    GenServer.start_link(__MODULE__, :ok, [])
  end

  @doc """
  Looks up the bucket pid for `name` stored in `server`.

  Returns `{:ok, pid}` if the bucket exists, `:error` otherwise.
  """
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end

  @doc """
  Ensures there is a bucket associated to the given `name` in `server`.
  """
  def create(server, name) do
    GenServer.cast(server, {:create, name})
  end

  ## Server Callbacks

  def init(:ok) do
    {:ok, %{}}
  end

  def handle_call({:lookup, name}, _from, names) do
    {:reply, Map.fetch(names, name), names}
  end

  def handle_cast({:create, name}, names) do
    if Map.has_key?(names, name) do
      {:noreply, names}
    else
      {:ok, bucket} = KV.Bucket.start_link()
      {:noreply, Map.put(names, name, bucket)}
    end
  end
end
```

第一个函数是 `start_link/0` ，它传递三个参数启动了一个新的GenServer：

1. 实现了服务器回调函数的模块名称。这里的 `__MODULE__` 指的是当前模块
2. 初始参数，这里是 `:ok`
3. 一组选项列表，比如可以存放服务器的名字。这里用个空列表

你可以向一个GenServer发送两种请求：`call` 和 `cast` 。**Call** 是同步的， 服务器 必须 发送回复给该类请求。**Cast** 是异步的，服务器 不会 发送回复消息。

再往下的两个方法， `lookup/2` 和 `create/2` ，它们用来发送这些请求给服务器。 这两种请求，会被第一个参数所指认的服务器中的 `handle_call/3` 和 `handle_cast/2` 函数处理（因此你的服务器回调函数必须包含这两个函数）。 `GenServer.call/2` 和 `GenServer.cast/2` 除了指认服务器之外，还告诉服务器它们要发送的请求。 这个请求存储在元组里，这里即 `{:lookup, name}` 和 `{:create, name}` ， 在下面写相应的回调处理函数时会用到。 这个消息元组第一个元素一般是要服务器做的事儿，后面的元素就是该动作的参数。

在服务器这边，我们要实现一系列服务器回调函数来实现服务器的启动、停止以及处理请求等。 回调函数是可选的，我们在这里只实现所关心的那几个。

第一个是 `init/1` 回调函数，它接受一个状态参数（你在用户API中调用 `GenServer.start_link/3` 中使用的那个）， 返回 `{:ok, state}` 。这里 `state` 是一个新建的图map。 我们现在已经可以观察到，GenServer的API中，客户端和服务器之间的界限十分明显。 `start_link/3` 在客户端发生。 而其对应的 `init/1` 在服务器端运行。

对于 `call` 请求，我们在服务器端必须实现 `handle_call/3` 回调函数。 参数：接收某请求（那个元组）、请求来源(`_from`)以及当前服务器状态（ `names` ）。 `handle_call/3` 函数返回一个 `{:reply, reply, new_state}` 元组。 其中， `reply` 是你要回复给客户端的东西，而 `new_statue` 是新的服务器状态。

对于 `cast` 请求，我们必须实现一个 `handle_cast/2` 回调函数， 接受参数： `request` 以及当前服务器状态（ `names` ）。 这个函数返回 `{:noreply, new_state}` 形式的元组。

这两个回调函数， `handle_call/3` 和 `handle_cast/2` 还可以返回其它几种形式的元组。 还有另外几种回调函数，如 `terminate/2` 和 `code_change/3` 等。 可以参考完整的GenServer文档来学习相关知识。

现在，来写几个测试来保证我们这个GenServer可以执行预期工作。

3.2-测试一个GenServer

测试一个GenServer比起测试agent没有多少区别。我们在测试的setup回调中启动该服务器进程用以测试。 用以下内容创建测试文件 `test/kv/registry_test.exs` ：

```
defmodule KV.RegistryTest do
  use ExUnit.Case, async: true

  setup do
    {:ok, registry} = KV.Registry.start_link
    {:ok, registry: registry}
  end

  test "spawns buckets", %{registry: registry} do
    assert KV.Registry.lookup(registry, "shopping") == :error

    KV.Registry.create(registry, "shopping")
    assert {:ok, bucket} = KV.Registry.lookup(registry, "shopping")

    KV.Bucket.put(bucket, "milk", 1)
    assert KV.Bucket.get(bucket, "milk") == 1
  end
end
```

哈，居然都过了！

我们不用显式关闭注册表进程，因为在测试执行完的时候它会自动收到 `:shutdown` 信号。 这个方法对于测试是还好啦。如果你想在GenServer的处理逻辑里加上关于停止的方法， 我们可以使用 `GenServer.stop/1` 函数：

```
## Client API

@doc """
Stops the registry.
"""
def stop(server) do
  GenServer.stop(server)
end
```

3.3-监控需求

至此我们的注册表完成的差不多了，剩下的问题就要解决在有bucket崩溃的时候注册表失去时效的问题。 比如给 `KV.RegistryTest` 增加一个测试来暴露这个问题：

```
test "removes buckets on exit", %{registry: registry} do
  KV.Registry.create(registry, "shopping")
  {:ok, bucket} = KV.Registry.lookup(registry, "shopping")
  Agent.stop(bucket)
  assert KV.Registry.lookup(registry, "shopping") == :error
end
```

这个测试会在最后一个断言处失败。因为当我们停止了bucket进程后，该bucket名字还存在于注册表中。

为了解决这个bug，我们需要注册表能够监视它派生出的每一个bucket进程。 一旦我们创建了监视器，注册表将收到每个bucket退出的通知。 这样它就可以清理bucket映射字典了。

我们先在命令行中玩弄一下监视机制。启动 `iex -S mix` ：

```
iex> {:ok, pid} = KV.Bucket.start_link
{:ok, #PID<0.66.0>}
iex> Process.monitor(pid)
#Reference<0.0.0.551>
iex> Agent.stop(pid)
:ok
iex> flush()
{:DOWN, #Reference<0.0.0.551>, :process, #PID<0.66.0>, :normal}
```

注意 `Process.monitor(pid)` 返回一个唯一的引用，使我们可以通过这个引用找到其指代的监视器发来的消息。 在我们停止agent之后，我们可以用 `flush()` 函数刷新所有消息， 此时会收到一个 `:DOWN` 消息，内含一个监视器返回的引用。它表示有个bucket进程退出， 原因是 `:normal` 。

现在让我们重新实现下服务器回调函数。

首先，将GenServer的状态改成两个字典：一个用来存储 `name->pid` 映射关系，另一个存储 `ref->name` 关系。 然后在 `handle_cast/2` 中加入监视器，并且实现一个 `handle_info/2` 回调函数用来保存监视消息。 下面是修改后完整的服务器调用函数：

```
## Server callbacks

def init(:ok) do
  names = %{}
  refs  = %{}
  {:ok, {names, refs}}
end

def handle_call({:lookup, name}, _from, {names, _} = state) do
  {:reply, Map.fetch(names, name), state}
end

def handle_cast({:create, name}, {names, refs}) do
  if Map.has_key?(names, name) do
    {:noreply, {names, refs}}
  else
    {:ok, pid} = KV.Bucket.start_link()
    ref = Process.monitor(pid)
    refs = Map.put(refs, ref, name)
    names = Map.put(names, name, pid)
    {:noreply, {names, refs}}
  end
end

def handle_info({:DOWN, ref, :process, _pid, _reason}, {names, refs}) do
  {name, refs} = Map.pop(refs, ref)
  names = Map.delete(names, name)
  {:noreply, {names, refs}}
end

def handle_info(_msg, state) do
  {:noreply, state}
end
```

看得出来，我们在没有修改客户端API情况下修改了服务器的实现。 这就体现出了GenServer将客户端与服务器隔离开的好处。

最后，不同于其他回调函数，我们定义了一个“捕捉所有消息”的 `handle_info/2` 的函数子句（可参考《入门》，其意类似重载的函数的一条实现）。它丢弃那些不知道也用不着的消息。 下面一节来解释下为啥。

3.4-call，cast还是info？

到目前为止，我们已经使用了三个服务器回调函数： `handle_call/3` ， `handle_cast/2` 和 `handle_info/2` 。何时使用哪个，其实很直白：

1. `handle_call/3` 用来处理同步请求。 这是默认的处理方式，因为等待服务器回复是十分有用的“压力反转（backpressure，涉及IO优化，请自行搜索）”机制。
2. `handle_cast/2` 用来处理异步请求，当你无所谓要不要个回复时。 一个cast请求甚至不保证服务器收到了该请求，因此请有节制地使用。 例如，我们定义的 `create/2` 函数应该使用call的，而我们用cast只是为了演示目的。
3. `handle_info/2` 用来接收和处理服务器收到的其它 （既不是 `GenServer.call/3` 也不是 `GenServer.cast/2` ）请求。 它可以接受是以普通进程身份通过 `send/2` 收到的消息或者其它消息。监视器发来的 `:DOWN` 消息就是个极好的例子。

因为任何消息，包括通过 `send/2` 发送的消息，回去到 `handle_info/2` 处理， 因此便会有很多你不需要的消息跑进服务器。如果不定义一个“捕捉所有消息”的函数子句， 这些消息会导致我们的监督者进程（supervisor）崩溃，因为没有函数子句匹配它们。

我们不需要为 `handle_call/3` 和 `handle_cast/2` 担心这个情况， 因为它们能接受的请求都是通过GenServer的API发送的，要是出了毛病就是程序员自己犯错。

3.5-监视器还是链接？

我们之前在 进程 那章里的学习过链接（links）。现在，随着注册表的完工， 你也许会问：我们啥时候用监控器，啥时候用链接呢？

链接是双向的。你将两个进程链接起来，其中一个挂了，另一个也会挂（除非它处理了该异常，改变了行为）。 而监视机制是单向的：只有监视别人的进程会收到被监视的进程的消息。 简单说，当你想让某些进程一挂都挂时，使用链接；而想要得到进程退出或挂了等事件的消息通知，使用监视。

回到我们 `handle_cast/2` 的实现，你可以看到注册表是同时链接着且监视着派生出的bucket：

```
{:ok, pid} = KV.Bucket.start_link()
ref = Process.monitor(pid)
```

这是个坏主意。我们不想注册表进程因为某个bucket进程挂而一同挂掉！ 我们将在讲解监督者（supervisor）时探索更好的解决方法。 一句话概括，我们将不直接创建新的进程，而是将把这个责任委托给监督者。 就像我们即将看到的那样，监督者同链接工作在一起，这就解释了为啥基于链接的API （如 `spawn_link` ， `start_link` 等）在Elixir和OTP上十分流行。

在讲监督者之前，我们首先探索下使用GenEvent进行事件管理以和处理的知识。

4-GenEvent

> 注：Elixir v1.1 发布后本章内容被从官方入门手册中拿掉了。 这里留存，如果仍需使用GenEvent，可以查阅。大家可以暂时跳过这一章。

本章探索GenEvent，Elixir和OTP提供的又一个行为抽象。它允许我们派生一个事件管理器，用来向多个处理者发布事件消息。

我们会激发两种事件：一个是每次bucket被加到注册表，另一个是从注册表中移除。

4.1-事件管理器

打开一个新 `iex -S mix` 对话，玩弄一下GenEvent的API：

```
iex> {:ok, manager} = GenEvent.start_link
{:ok, #PID<0.83.0>}
iex> GenEvent.sync_notify(manager, :hello)
:ok
iex> GenEvent.notify(manager, :world)
:ok
```

函数 `GenEvent.start_link/0` 启动了一个新的事件管理器。不需额外的参数。 管理器创建好后，我们就可以调用 `GenEvent.notify/2` 函数和 `GenEvent.sync_notify/2` 函数来发送通知。

但是，当前还没有任何消息处理者绑定到该管理器，因此不管它发啥通知，叫破喉咙都不会有事儿发生。

现在就在iex对话里创建第一个事件处理器：

```
iex> defmodule Forwarder do
...>   use GenEvent
...>   def handle_event(event, parent) do
...>     send parent, event
...>     {:ok, parent}
...>   end
...> end
iex> GenEvent.add_handler(manager, Forwarder, self())
:ok
iex> GenEvent.sync_notify(manager, {:hello, :world})
:ok
iex> flush
{:hello, :world}
:ok
```

我们创建了一个处理器（handler），并通过函数`GenEvent.add_handler/3`把它“绑定”到事件管理器上，传递的三个参数是：

1. 刚启动的那个时间管理器
2. 定义事件处理者的模块（如这里的`Forwarder`）
3. 事件处理者的状态：在这里，使用当前进程的id

加上这个处理器之后，可以看到，调用了`sync_notify/2`之后，`Forwarder`处理器成功地把事件转给了它的父进程（IEx），因此那个消息进入了我们的收件箱。

这里有几点需要注意：

1. 事件处理器运行在事件管理器的同一个进程里
2. `sync_notify/2`同步地运行事件处理器处理请求
3. `notify/2`使事件处理器异步处理请求

这里`sync_notify/2`和`notify/2`类似于GenServer里面的`call/2`和`cast/2`。推荐使用`sync_notify/2`。 它以反向压力的机制工作，减少了“发消息速度快过消息被成功分发的速度”的可能性。

记得去GenServer的模块文档阅读其它函数。 目前我们的程序就用提到的这些知识就可以了。

4.2-注册表进程的事件

为了能发出事件消息，我们要稍微修改一下我们的注册表进程，使之与一个事件管理器进行协作。 我们需要在注册表进程启动的时候，事件管理器也能自动启动。 比如在`init/1`回调里面，最好能传递事件处理器的pid或名字什么的作为参数来`start_link`，以此将启动事件管理器与注册表进程分解开。

但是，首先让我们修改测试中注册表进程的行为。打开`test/kv/registry_text.exs`，修改目前的`setup`回调，然后再加上新的测试：

```
defmodule Forwarder do
  use GenEvent

  def handle_event(event, parent) do
    send parent, event
    {:ok, parent}
  end
end

setup do
  {:ok, manager} = GenEvent.start_link
  {:ok, registry} = KV.Registry.start_link(manager)

  GenEvent.add_mon_handler(manager, Forwarder, self())
  {:ok, registry: registry}
end

test "sends events on create and crash", %{registry: registry} do
  KV.Registry.create(registry, "shopping")
  {:ok, bucket} = KV.Registry.lookup(registry, "shopping")
  assert_receive {:create, "shopping", ^bucket}

  Agent.stop(bucket)
  assert_receive {:exit, "shopping", ^bucket}
end
```

为了测试我们即将添加的功能，我们首先定义了一个 `Forwarder` 事件处理器，类似刚才在IEx中创建的那样。 在 `Setup` 中，我们启动了事件管理器，把它作为参数传递给了注册表进程，并且向该管理器添加了我们定义的 `Forwarder` 处理器。 至此，事件可以发向待测进程了。

在测试中，我们创建、停止了一个bucket进程，并且使用 `assert_receive` 断言来检查是否收到了 `:create` 和 `:exit` 事件消息。 断言 `assert_receive` 默认是500毫秒超时时间，这对于测试足够了。 同样要指出的是， `assert_receive` 期待接收一个模式，而不是一个值。 这就是为啥我们用 `^bucket` 来匹配bucket的pid（参考《入门》关于变量的匹配内容）。

最终，注意我们调用了 `GenEvent.add_mon_handler/3` 来代替 `GenEvent.add_handler/3` 。该函数不但可以添加一个处理器，它还告诉事件管理器来监视当前进程。如果当前进程挂了，事件处理器也一并抹去。 这个很有道理，因为对于这里的 `Forwarder` ，如果消息的接收方（ `self()` /测试进程）终止，我们理所应当停止转发消息。

好了，现在来修改注册表进程代码来让测试pass。打开 `lib/kv/registry.ex` ，输入以下新的内容（一些关键语句的解释写在注释里）：

```
defmodule KV.Registry do
  use GenServer

  ## Client API

  @doc """
  Starts the registry.
  """
  def start_link(event_manager, opts \\ []) do
    # 1. start_link now expects the event manager as argument
    GenServer.start_link(__MODULE__, event_manager, opts)
  end

  @doc """
  Looks up the bucket pid for `name` stored in `server`.
```

```
  Returns `{:ok, pid}` in case a bucket exists, `:error` otherwise.
  """
  def lookup(server, name) do
    GenServer.call(server, {:lookup, name})
  end

  @doc """
  Ensures there is a bucket associated with the given `name` in `server`.
  """
  def create(server, name) do
    GenServer.cast(server, {:create, name})
  end

  ## Server callbacks

  def init(events) do
    # 2. The init callback now receives the event manager.
    #    We have also changed the manager state from a tuple
    #    to a map, allowing us to add new fields in the future
    #    without needing to rewrite all callbacks.
    names = HashDict.new
    refs  = HashDict.new
    {:ok, %{names: names, refs: refs, events: events}}
  end

  def handle_call({:lookup, name}, _from, state) do
    {:reply, HashDict.fetch(state.names, name), state}
  end

  def handle_cast({:create, name}, state) do
    if HashDict.get(state.names, name) do
      {:noreply, state}
    else
      {:ok, pid} = KV.Bucket.start_link()
      ref = Process.monitor(pid)
      refs = HashDict.put(state.refs, ref, name)
      names = HashDict.put(state.names, name, pid)
      # 3. Push a notification to the event manager on create
      GenEvent.sync_notify(state.events, {:create, name, pid})
      {:noreply, %{state | names: names, refs: refs}}
    end
  end

  def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
    {name, refs} = HashDict.pop(state.refs, ref)
    names = HashDict.delete(state.names, name)
    # 4. Push a notification to the event manager on exit
    GenEvent.sync_notify(state.events, {:exit, name, pid})
    {:noreply, %{state | names: names, refs: refs}}
  end

  def handle_info(_msg, state) do
    {:noreply, state}
  end
end
```

这些改变很直观。我们给 `GenServer` 初始化过程传递一个事件管理器，该管理器是我们用 `start_link` 启动进程时作为参数收到的。 我们还改了cast和info两个回调，在里面调用了 `GenEvent.sync_notify/2` 。 最后，我们借这个机会还把服务器的状态改成了一个图，方便我们以后改进注册表进程。

执行测试，都是绿的。

4.3-事件流

最后一个值得探索的`GenEvent`的功能点是像处理流一样处理事件：

```
iex> {:ok, manager} = GenEvent.start_link
{:ok, #PID<0.83.0>}
iex> spawn_link fn ->
...>   for x <- GenEvent.stream(manager), do: IO.inspect(x)
...> end
:ok
iex> GenEvent.notify(manager, {:hello, :world})
{:hello, :world}
:ok
```

上面的例子中，我们创建了一个`GenEvent.stream(manager)`，返回一个事件的流（即一个enumerable），并随即处理了它。 处理事件是一个阻塞的行为，我们派生新进程来处理事件消息，把消息打印在终端上。这一系列的操作，就像看到的那样，如实地执行了。 每次调用`sync_notify/2`或者`notify/2`，事件都被打印在终端上，后面跟着一个`:ok`（IEx输出语句的执行结果）。

通常事件流提供了足够多的内置功能来处理事件，使我们不必实现我们自己的处理器。 但是，若是需要某些自定义的功能，或是在测试时，定义自己的事件处理器回调才是正道。

至此，我们有了一个事件处理器，一个注册表进程以及可能会同时执行的许多bucket进程，是时候开始担心这些进程会不会挂掉了。

5-监督者和应用程序

到目前为止，我们的程序已经实现了注册表（registry）来对成百上千的bucket进程进行监视。 你是不是觉得这个还不错？没有软件是bug-free的，挂掉那是必定会发生滴。

当有东西挂了，我们的第一反应是："快拯救这些错误"。但是，像在《入门》中学到的那样，不同于其它多数语言，Elixir不太做"防御性编程"。 相反，我们说"要挂快点挂"，或是"就让它挂"。 如果有bug要让我们的注册表进程挂掉，啥也别怕，因为我们即将实现用监督者来启动新的注册表进程副本。

本章我们将学习监督者（supervisor），还会讲到些有关应用程序的知识。 一个不够，我们要创建两个监督者，用它们监督我们的进程。

5.1-第一个监督者

创建一个监督者跟创建通用服务器差不多。我们将定义一个名为 `KV.Supervisor` 的模块， 使用Supervisor行为。 代码文件 `lib/kv/supervisor.ex` 内容如下：

```
defmodule KV.Supervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, :ok)
  end

  def init(:ok) do
    children = [
      worker(KV.Registry, [KV.Registry])
    ]

    supervise(children, strategy: :one_for_one)
  end
end
```

我们的监督者目前只有一个孩子：注册表进程。一个形式如

```
worker(KV.Registry, [KV.Registry])
```

的worker，在调用：

```
KV.Registry.start_link(KV.Registry)
```

时将启动一个进程。

我们传给 `start_link` 的参数是进程的名称。给监督机制下得进程命名是常见的做法， 这样别的进程就可以通过名称访问它们，而不需要知道它们的进程ID。 这很有用，因为当被监督的某进程挂掉被重启后，它的进程ID可能会改变。但是用名称就不一样了。 我们可以保证一个挂掉新启的进程，还会用同样的名称注册进来。而不用显式地先获取之前的进程ID。 另外，通常会用定义的模块名称作为进程的名字，在将来对系统进行debug时非常直观。

最后，我们调用了 `supervisor/2` ，给它传递了一个孩子列表以及策略： `:one_for_one` 。

监督者的策略指明了当一个孩子进程挂了会发生什么。 `:one_for_one` 意思是如果一个孩子进程挂了， 只有一个“复制品”会启动来替代它。我们现在要的就是这个策略， 因为我们只有一个孩子。 `Supervisor` 支持许多不同的策略，我们在本章中将会陆续讨论。

因为 `KV.Registry.start_link/1` 现在期待一个参数，需要修改我们的实现来接受这一个参数。打开文件 `lib/kv/registry.ex` ，覆盖原来的 `start_link/0` 定义：

```
@doc """
Starts the registry with the given `name`.
"""
def start_link(name) do
  GenServer.start_link(__MODULE__, :ok, name: name)
end
```

我们还要修改测试，在注册表进程启动时给个名字。在文件 `test/kv/registry_test.exs` 中覆盖原 `setup` 函数代码：

```
setup context do
  {:ok, registry} = KV.Registry.start_link(context.test)
  {:ok, registry: registry}
end
```

类似 `test/3` ，函数 `setup/2` 也接受测试上下文（context）。不管我们给setup代码中添加了啥， 上下文中包含着几个关键变量：比如 `:case` ， `:test` ， `:file` 和 `:line` 。 上面代码中，我们用了 `context.test` 作为捷径取得当前运行着的测试名称，生成一个注册表进程。

现在，随着测试通过，可以拉我们的监督者出去溜溜了。如果在工程中启动命令行对话 `iex -S mix` ， 我们可以手动启动监督者：

```
iex> KV.Supervisor.start_link
{:ok, #PID<0.66.0>}
iex> KV.Registry.create(KV.Registry, "shopping")
:ok
iex> KV.Registry.lookup(KV.Registry, "shopping")
{:ok, #PID<0.70.0>}
```

当我们启动监督者，注册表worker会自动启动，允许我们创建bucket而不需要手动启动它们。

但是，在实战中我们很少手动启动应用程序的监督者。启动监督者是应用程序回调过程的一部分。

5.2-理解应用程序

起始我们已经把所有时间都花在这个应用程序上了。每次修改了一个文件，执行 `mix compile` ，我们都能看到 `Generated kv app` 消息在编译信息中打印出来。

我们可以在 `_build/dev/lib/kv/ebin/kv.app` 找到 `.app` 文件。来看一下它的内容：

```
{application,kv,
             [{registered,[]},
              {description,"kv"},
              {applications,[kernel,stdlib,elixir,logger]},
              {vsn,"0.0.1"},
              {modules,['Elixir.KV','Elixir.KV.Bucket',
                        'Elixir.KV.Registry','Elixir.KV.Supervisor']}]}.
```

该文件包含Erlang的语句（使用Erlang的语法写的）。但即使我们不熟悉Erlang，也能很容易地猜到这个文件保存的是我们应用程序的定义。 它包括应用程序的版本，定义的所有模块，还有它依赖的应用程序列表， 如Erlang的Kernel，elixir本身，logger（我们在 `mix.exs` 里添加的）。

要是每次我们添加一个新的模块就要手动修改这个文件，是很讨厌的。 这也是为啥把它交给mix来自动维护的原因。

我们还可以通过修改 `mix.exs` 工程文件中，函数 `application/0` 的返回值， 来配置生成的 `.app` 文件。我们将很快做第一次自定义配置。

5.2.1-启动应用程序

定义了 `.app` 文件（里面是应用程序的定义），我们就可以将应用程序视作一个整体形式来启动和停止。 到目前为止我们还没有考虑过这个问题，这是因为：

1. Mix为我们自动启动了应用程序
2. 即使Mix没有自动启动我们的程序，该程序启动后也没做啥特别的事儿

总之，让我们看看Mix如何为我们启动应用程序。先在工程下启动命令行，然后试着执行：

```
iex> Application.start(:kv)
{:error, {:already_started, :kv}}
```

擦，已经启动了？Mix通常会启动文件 `mix.exs` 中定义的整个应用程序结构。 遇到依赖的程序也会如此一并启动。

我们可以给mix一个选项，让它不要启动我们的应用程序。 执行命令： `iex -S mix run --no-start` 启动命令行，然后执行：

```
iex> Application.start(:kv)
:ok
```

我们可以停止 `:kv` 程序和 `:logger` 程序，后者是Elixir默认情况下自动启动的：

```
iex> Application.stop(:kv)
:ok
iex> Application.stop(:logger)
:ok
```

然后再次启动我们的程序：

```
iex> Application.start(:kv)
{:error, {:not_started, :logger}}
```

错误是由于 `:kv` 所依赖的应用程序（这里是 `:logger` ）没有启动导致的。 Mix一般会根据工程中的 `mix.exs` 启动整个应用程序结构； 对其依赖的每个应用程序来说也是这样（如果它们还依赖于其它应用程序）。 但是这次我们用了 `--no-start` 标志，因此我们需要手动 按顺序启动所有应用程序， 或者像这样调用 `Application.ensure_all_started` ：

```
iex> Application.ensure_all_started(:kv)
{:ok, [:logger, :kv]}
```

没什么激动人心的，这些只是演示了如何控制我们的应用程序。

> 当你运行 `iex -S mix` ，它相当于执行 `iex -S mix run` 。 因此无论何时你启动iex会话，传递参数给 `mix run` ，实际上是传递给 `run` 命令。 你可以在命令行中执行 `mix help run` 获取关于 `run` 的更多信息。

5.2.2-应用程序的回调（callback）

因为我们几乎都在讲应用程序如何启动和停止，你能猜到肯定有办法能在启动的当儿做点有意义的事情。 没错，有的！

我们可以定义应用程序的回调函数。在应用程序启动时，该函数将被调用。 这个函数必须返回 `{:ok, pid}` ，其中 `pid` 是其内部监督者进程的标识符。

我们分两步来定义这个回调函数。首先，打开 `mix.exs` 文件，修改 `def application` 部分：

```
def application do
  [applications: [:logger],
   mod: {KV, []}]
end
```

选项 `:mod` 指出了“应用程序回调函数的模块”，后面跟着该传递给它的参数。 这个回调函数的模块可以是任意模块，只要它实现了Application行为。

在这里，我们要让 `KV` 作为它回调函数的模块。因此在文件 `lib/kv.ex` 中做一些修改：

```
defmodule KV do
  use Application

  def start(_type, _args) do
    KV.Supervisor.start_link
  end
end
```

当我们声明 `use Application` ，（类似声明了 `GenServer` 、 `Supervisor` ） 我们需要定义几个函数。这里我们只需定义 `start/2` 函数。 如果我们想在应用程序停止时定义一个自定义的行为，我们也可以定义一个 `stop/1` 函数。

现在我们再次用 `iex -S mix` 启动我们的工程对话。 我们将看到一个名为 `KV.Registry` 的进程已经在运行：

```
iex> KV.Registry.create(KV.Registry, "shopping")
:ok
iex> KV.Registry.lookup(KV.Registry, "shopping")
{:ok, #PID<0.88.0>}
```

好牛逼！

5.2.3-工程还是应用程序？

Mix是区分工程（projects）和应用程序（applications）的。 基于目前的 `mix.exs` ，我们可以说，我们有一个Mix 工程，该工程定义了 `:kv` 应用程序。 在后面章节我们会看到，有些工程一个应用程序也没定义。

当我们讲“工程”时，你应该想到Mix。Mix是管理工程的工具。 它知道如何去编译、测试你的工程，等等。它还知道如何编译和启动你的工程的相关应用程序。

当我们讲“应用程序”时，我们讨论的是OTP。应用程序是一个实体，它作为一个整体启动或者停止。 你可以在应用程序模块文档 阅读更多关于应用程序的知识。或者执行 `mix help compile.app` 来学习 `def application` 中支持的更多选项。

5.3 简单的一对一监督者

我们已经成功定义了我们的监督者，它作为我们应用程序生命周期的一部分自动启动（和停止）。

回顾一下，我们的 `KV.Registry` 在 `handle_cast/2` 回调中，链接并且监视bucket进程：

```
{:ok, pid} = KV.Bucket.start_link()
ref = Process.monitor(pid)
```

链接是双向的，意味着一个bucket进程挂了会导致注册表进程挂掉。 尽管现在我们有了监督者，它能保证一旦注册表进程挂了还可以重启。 但是注册表挂掉仍然意味着我们会丢失用来匹配bucket名称到其相应进程的数据。

换句话说，我们希望即使bucket进程挂了，注册表进程也能够保持运行。写成测试就是：

```
test "removes bucket on crash", %{registry: registry} do
    KV.Registry.create(registry, "shopping")
    {:ok, bucket} = KV.Registry.lookup(registry, "shopping")

    # Stop the bucket with non-normal reason
    Process.exit(bucket, :shutdown)

    # Wait until the bucket is dead
    ref = Process.monitor(bucket)
    assert_receive {:DOWN, ^ref, _, _, _}

    assert KV.Registry.lookup(registry, "shopping") == :error
  end
```

这个测试很像之前的“退出时移除bucket”， 只是我们的做法更加残暴（用 `:shutdown` 代替了 `:normal` ）。 不像 `Agent.stop/1` ， `Process.exit/2` 是一个异步的操作。 因此我们不能简单地在刚发了退出信号之后就执行查询 `KV.Registry.lookup/2` ， 那个时候也许bucket进程还没有结束（也就不会造成系统问题）。 为了解决这个问题，我们仍然要在测试期间监视bucket进程，然后在确保其已经结束时再去查询注册表进程， 避免竞争状态。

因为bucket是链接注册表进程的，而注册表进程是链接着测试进程。让bucket挂掉会导致测试进程挂掉：

```
1) test removes bucket on crash (KV.RegistryTest)
   test/kv/registry_test.exs:52
   ** (EXIT from #PID<0.94.0>) shutdown
```

一个可行的解决方法是提供 `KV.Bucket.start/0` ，让它执行 `Agent.start/1` 。 在注册表进程中使用这个方法启动bucket，从而避免它们之间的链接。 但是这不是个好办法，因为这样bucket进程就链接不到任何进程。 这意味着所有bucket进程即使在有人停止了 `:kv` 程序也一直活着。 不光如此，它的进程会变得不可触及。而一个不可触及的进程是难以在运行时内省的。

我们将定义一个新的监督者来解决这个问题。这个新监督者会派生和监督所有的bucket。 有一个简单的一对一监督策略，叫做 `:simple_one_for_one` ，对于此情况是非常适用的： 他允许指定一个工人模板，而后监督基于那个模板创建的多个孩子。 在这个策略下，工人进程不会在监督者初始化时启动。而是每次调用了 `start_child/2` 函数后， 才会创建一个新的工人进程。

让我们在文件 `lib/kv/bucket/supervisor.ex` 中定义 `KV.Bucket.Supervisor` ：

```
defmodule KV.Bucket.Supervisor do
  use Supervisor

  # A simple module attribute that stores the supervisor name
  @name KV.Bucket.Supervisor

  def start_link() do
    Supervisor.start_link(__MODULE__, :ok, name: @name)
  end

  def start_bucket do
    Supervisor.start_child(@name, [])
  end

  def init(:ok) do
    children = [
      worker(KV.Bucket, [], restart: :temporary)
    ]

    supervise(children, strategy: :simple_one_for_one)
  end
end
```

比起我们第一个监督者，这个监督者有三点改变。

相较于之前接受所注册进程的名字作为参数，我们这里只简单地将其命名为 `KV.Bucket.Supervisor` （代码中用 `__MODULE__` ），因为我们不需要派生这个进程的多个版本。

我们还定义了函数 `start_bucket/0` 来启动每个bucket，作为这个名为 `KV.Bucket.Supervisor` 的监督者的孩子。函数 `start_bucket/0` 代替了注册表进程中直接调用的 `KV.Bucket.start_link` 。

最后，在 `init/1` 回调中，我们将工人进程标记为 `:temporary` 。意思是如果bucket进程即使挂了也不回重启。因为我们创建这个监督者，只是用来作为将bucket进程圈成组这么一种机制。bucket进程的创建还应该通过注册表进程。

执行 `iex -S mix` 来试用下这个新监督者：

```
iex> {:ok, _} = KV.Bucket.Supervisor.start_link
{:ok, #PID<0.70.0>}
iex> {:ok, bucket} = KV.Bucket.Supervisor.start_bucket()
{:ok, #PID<0.72.0>}
iex> KV.Bucket.put(bucket, "eggs", 3)
:ok
iex> KV.Bucket.get(bucket, "eggs")
3
```

修改注册表进程中启动bucket的部分，来与bucket的监督者协同工作：

```
def handle_cast({:create, name}, {names, refs}) do
  if Map.has_key?(names, name) do
    {:noreply, {names, refs}}
  else
    {:ok, pid} = KV.Bucket.Supervisor.start_bucket()
    ref = Process.monitor(pid)
    refs = Map.put(refs, ref, name)
    names = Map.put(names, name, pid)
    {:noreply, {names, refs}}
  end
end
```

在做了这些修改之后，我们的测试还是会fail。因为bucket的监督者还没有启动。 但是我们将不会在每次测试启动时启动bucket的监督者，而是让其作为我们主监督者树的一部分自动启动。

5.4-监督树

为了在应用程序中使用bucket的监督者，我们要把它作为一个孩子加到 `KV.Supervisor` 中去。注意，我们已经开始用一个监督者去监督另一个监督者了---正式的称呼是“监督树”。

打开 `lib/kv/supervisor.ex` ，添加一个新的模块属性存储bucket监督者的名字， 并且修改 `init/1` ：

```
def init(:ok) do
   children = [
     worker(KV.Registry, [KV.Registry]),
     supervisor(KV.Bucket.Supervisor, [])
   ]

   supervise(children, strategy: :one_for_one)
 end
```

这里我们添加了一个监督者作为孩子（没有传递启动参数）。重新运行测试，测试将可以通过。

记住，声明各个孩子的顺序是很重要的。因为注册表进程依赖于bucket监督者， 所以bucket监督者需要在孩子列表中排得靠前一些。

因为我们已为监督者添加了多个孩子，现在就需要考虑使用 `:one_for_one` 这个策略还是否正确。 一个显现的问题就是注册表进程和bucket监督者之间的关系。 如果注册表进程挂了，bucket监督者也必须挂。 因为一旦注册表进程挂了，所有关联bucket名字和其进程的信息也就丢失了。 此时若bucket的监督者还活着，它掌管的众多bucket将根本访问不到，变成垃圾。

我们可以考虑使用其他的策略，如 `:one_for_all` 或 `:rest_for_one` 。 策略 `:one_for_all` 在任何时候，只要有一个孩子挂，它就会停止并且重启所有孩子进程。 这个貌似符合现在的需求，但是有些简单粗暴。因为如果bucket监督者进程挂了， 是没必要同时挂掉注册表进程

的。因为注册表进程本身就监控这每个bucket进程的状态， 它会自己清理不需要的信息（挂掉的bucket）。因此，策略 `:rest_for_one` 是比较合适的。 它会单独重启挂掉的孩子进程，而不影响其它的。因此我们做如下修改：

```
def init(:ok) do
  children = [
    worker(KV.Registry, [KV.Registry]),
    supervisor(KV.Bucket.Supervisor, [])
  ]

  supervise(children, strategy: :rest_for_one)
end
```

如果注册表进程挂了，那么它和bucket监督者都会被重启； 而如果只是bucket监督者进程挂了，那么只有它自己被重启。

还有其它几个策略或选项可以传递给 `worker/2` ， `supervisor/2` 和 `supervise/2` 函数， 所以可别忘记阅读监督者 及监督者.spec的文档。

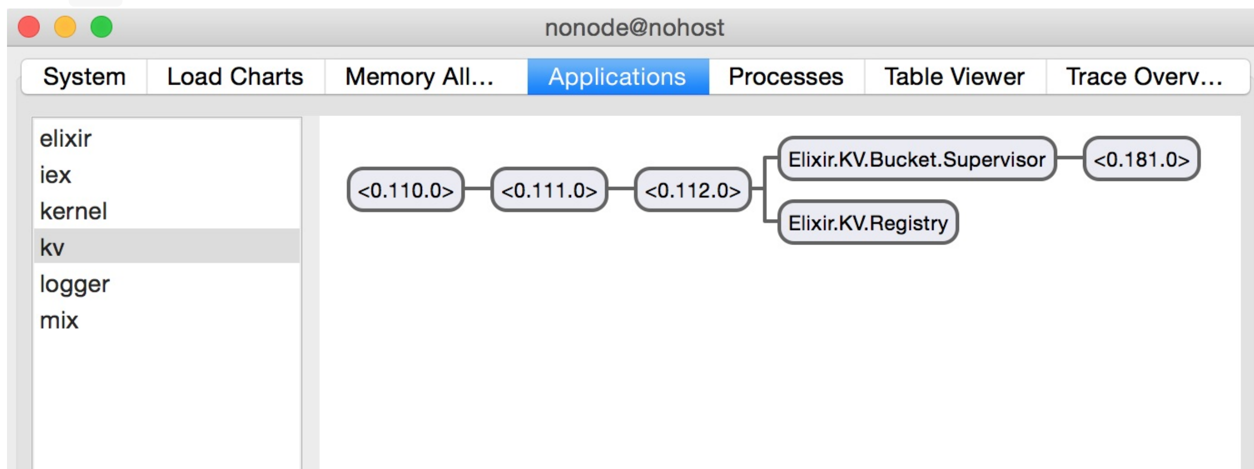
5.5 观察者（Observer）

现在我们定义好了监督者树，这是介绍观察者工具（Observer tool）的最佳时机。 该工具和Erlang一同推出。使用 `iex -S mix` 启动你的应用程序，输入：

```
iex> :observer.start
```

一个GUI窗口将弹出，里面包含了关于我们系统的各种信息：从总体统计信息到负载图表，还有运行中的所有进程和应用程序。

在“应用程序”Tab页上，可以看到系统中运行的所有应用程序以及它们的监督者树信息。 可以选择 `kv` 查看它的详细信息：



不但如此，如果你再命令行中创建新的bucket：

```
iex> KV.Registry.create KV.Registry, "shopping"
:ok
```

你可以在观察者工具中看到从监督者树种派生出了新的进程。

观察者工具就留给读者自行探索。你可以双击进程查看其详细信息， 还可以右击发送停止信号（模拟进程失败的完美方法）等等。

在每天辛苦工作快要结束的时候，一个像观察者这样的工具绝对是你还想着在监督者树里创建几条进程的主要原因之一。 即使创建的都是临时的，你也可以看看整个工程里各个进程还是不是可触及或是可内省的。

5.6 测试里共享的状态

目前为止，我们是在每个测试中启动一个注册表进程，以确保它们是独立的：

```
setup context do
  {:ok, registry} = KV.Registry.start_link(context.test)
  {:ok, registry: registry}
end
```

因为我们已经将注册表进程改成使用 `KV.Bucket.Supervisor` 了，而它是在全局注册的， 因此现在我们的测试依赖于这个共享的、全局的监督者，即使每个测试仍使用自己的注册表进程。 那么问题来了：我们是否应该这么做？

It depends。只要仅依赖于某一状态的非共享部分，那么也还ok啦。比如，每次用一个名字注册进程， 都是注册在一个共享的注册表中。尽管如此，只要确保每个名字用于不同的测试，比如在创建时使用上下文参数 `context.test` ，就不会再测试间出现并行或者数据依赖的问题。

对我们的bucket监督者来说也是同样的道理。尽管多个注册表进程会在共享的bucket监督者上启动bucket， 但这些bucket和注册表进程之间是相互隔离的。我们唯一会遇到并发问题，是我们想要调用 函数 `Supervisor.count_children(KV.Bucket.Supervisor)` 的时候。 它统计所有注册表进程下的所有bucket。当测试并行执行并调用它的时候，返回的结果可能不一样。

因此，目前由于我们的测试依赖于共享的监督者中的非共享部分，我们不用担心并发问题。假如它成为问题了，我们可以给每个测试启动一个监督者，并将其作为参数传递给注册表进程的 `start_link` 函数。

至此，我们的应用程序已经被监督者监督着，而且也已测试通过。之后我们要想办法提升一些性能。

6-ETS

每次我们要找一个bucket时，都要发消息给注册表进程。在某些情况下，这意味着注册表进程会变成性能瓶颈！

本章我们将学习ETS（Erlang Term Storage），以及如何把它当成缓存使用。 之后我们会拓展它的功能，把数据从监督者保存到其孩子上。这样即使崩溃，数据也能存续。

> 严重注意！绝对不要冒失地把ETS当缓存用。仔细分析你的程序，看看到底哪里才是瓶颈。这样来决定是否需要缓存以及缓存什么。 本章仅仅讲解ETS是如何工作的一个例子，具体怎么做得由你自己决定。

6.1-ETS当缓存用

ETS可以把Erlang/Elixir的词语（term）存储在内存表中。 使用Erlang的 `:ets` 模块来操作：

```
iex> table = :ets.new(:buckets_registry, [:set, :protected])
8207
iex> :ets.insert(table, {"foo", self})
true
iex> :ets.lookup(table, "foo")
[{"foo", #PID<0.41.0>}]
```

在创建一个ETS表时，需要两个参数：表名和一组选项。对于在上面的例子，在可选的选项中我们传递了表类型和访问规则。 我们选择了 `:set` 类型，意思是键不能有重复（集合论）。 我们选择的访问规则是 `:protected` ，意思是对于这个表，只有创建该表的进程可以修改，而其它进程只能读取。 这两个选项是默认的，这里就不多说了。

ETS表可以被命名，可以通过名字访问：

```
iex> :ets.new(:buckets_registry, [:named_table])
:buckets_registry
iex> :ets.insert(:buckets_registry, {"foo", self})
true
iex> :ets.lookup(:buckets_registry, "foo")
[{"foo", #PID<0.41.0>}]
```

好了，现在我们使用ETS表，修改 `KV.Registry` 。 我们对事件管理器和bucket的监督者使用相同的技术，显式传递ETS表名给 `start_link` 。 记住，有了服务器以及ETS表的名字，本地进程就可以访问那个表。

打开 `lib/kv/registry.ex` ，修改里面的实现。加上注释来标明我们的修改：

```
defmodule KV.Registry do
  use GenServer

  ## Client API

  @doc """
  Starts the registry.
  """
  def start_link(table, event_manager, buckets, opts \\ []) do
    # 1. We now expect the table as argument and pass it to the server
    GenServer.start_link(__MODULE__, {table, event_manager, buckets}, opts)
  end

  @doc """
  Looks up the bucket pid for `name` stored in `table`.

  Returns `{:ok, pid}` if a bucket exists, `:error` otherwise.
  """
  def lookup(table, name) do
    # 2. lookup now expects a table and looks directly into ETS.
    #    No request is sent to the server.
    case :ets.lookup(table, name) do
      [{^name, bucket}] -> {:ok, bucket}
      [] -> :error
    end
  end

  @doc """
  Ensures there is a bucket associated with the given `name` in `server`.
  """
  def create(server, name) do
    GenServer.cast(server, {:create, name})
  end

  ## Server callbacks

  def init({table, events, buckets}) do
    # 3. We have replaced the names HashDict by the ETS table
    ets  = :ets.new(table, [:named_table, read_concurrency: true])
    refs = HashDict.new
    {:ok, %{names: ets, refs: refs, events: events, buckets: buckets}}
  end

  # 4. The previous handle_call callback for lookup was removed

  def handle_cast({:create, name}, state) do
    # 5. Read and write to the ETS table instead of the HashDict
    case lookup(state.names, name) do
      {:ok, _pid} ->
        {:noreply, state}
      :error ->
        {:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
        ref = Process.monitor(pid)
        refs = HashDict.put(state.refs, ref, name)
        :ets.insert(state.names, {name, pid})
        GenEvent.sync_notify(state.events, {:create, name, pid})
        {:noreply, %{state | refs: refs}}
    end
  end

  def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
    # 6. Delete from the ETS table instead of the HashDict
    {name, refs} = HashDict.pop(state.refs, ref)
    :ets.delete(state.names, name)
    GenEvent.sync_notify(state.events, {:exit, name, pid})
    {:noreply, %{state | refs: refs}}
  end
```

```
  def handle_info(_msg, state) do
    {:noreply, state}
  end
end
```

注意，修改前的 `KV.Registry.lookup/2` 给服务器发送请求；修改后，它就直接从ETS表里面读取数据了。该表是对各进程都共享的。 这就是我们实现的缓存机制的大体想法。

为了让缓存机制工作，新建的ETS起码需要 `:protected` 访问规则（默认的），这样客户端才能从中读取数据。 否则就只有 `KV.Registry` 进程才能访问。 我们还在启动ETS表时设置了 `:read_concurrency` ，为表的并发访问稍作优化。

我们以上的改动导致测试都挂了。一个重要原因是我们在启动注册表进程时，需要多传递一个参数给 `KV.Registry.start_link/3` 。 让我们重写 `setup` 回调来修复测试代码 `test/kv/registry_test.exs` ：

```
setup do
  {:ok, sup} = KV.Bucket.Supervisor.start_link
  {:ok, manager} = GenEvent.start_link
  {:ok, registry} = KV.Registry.start_link(:registry_table, manager, sup)

  GenEvent.add_mon_handler(manager, Forwarder, self())
  {:ok, registry: registry, ets: :registry_table}
end
```

注意我们传递了一个表名 `:registry_table` 给 `KV.Registry.start_link/3` ， 其后返回了 `ets: :registry_table` ，成为了测试的上下文。

修改了这个回调后，测试仍有fail，差不多都是这个样子：

```
1) test spawns buckets (KV.RegistryTest)
   test/kv/registry_test.exs:38
   ** (ArgumentError) argument error
   stacktrace:
     (stdlib) :ets.lookup(#PID<0.99.0>, "shopping")
     (kv) lib/kv/registry.ex:22: KV.Registry.lookup/2
     test/kv/registry_test.exs:39
```

这是因为我们传递了注册表进程的pid给函数 `KV.Registry.lookup/2` ，而它期待的却是ETS的表名。 为了修复我们要把所有的：

```
KV.Registry.lookup(registry, ...)
```

都改为：

```
KV.Registry.lookup(ets, ...)
```

其中获取 `ets` 的方法跟我们获取注册表一个样子：

```
test "spawns buckets", %{registry: registry, ets: ets} do
```

像这样，我们对测试进行修改，把 `ets` 传递给 `lookup/2` 。一旦我们完成这些修改，有些测试还是会失败。 你还会观察到，每次执行测试，成功和失败不是稳定的。例如，对于“派生bucket进程”这个测试来说：

```
test "spawns buckets", %{registry: registry, ets: ets} do
  assert KV.Registry.lookup(ets, "shopping") == :error

  KV.Registry.create(registry, "shopping")
  assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")

  KV.Bucket.put(bucket, "milk", 1)
  assert KV.Bucket.get(bucket, "milk") == 1
end
```

有可能会在这行失败：

```
assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

但是假如我们在这行之前创建一个bucket，还会失败吗？

原因在于（嗯哼！基于教学目的），我们犯了两个错误：

1. 我们过于冒进地使用缓存来优化
2. 我们使用的是 `cast/2` ，它应该是 `call/2`

6.2-竞争条件？

用Elixir编程不会让你避免竞争状态。但是Elixir关于“没啥是共享”的这个特点可以帮助你很容易找到导致竞争状态的根本原因。

我们测试中发生的事儿是延迟---介于我们操作和我们观察到ETS表被改动之间。下面是我们期望发生的：

1. 我们执行 `KV.Registry.create(registry, "shopping")`
2. 注册表进程创建了bucket，并且更新了缓存表
3. 我们用 `KV.Registry.lookup(ets, "shopping")` 从表中获取信息
4. 上面的命令返回 `{:ok, bucket}`

但是，因为 `KV.Registry.create/2` 使用cast操作，命令在真正修改表之前先返回了结果！换句话说，其实发生了下面的事：

1. 我们执行 `KV.Registry.create(registry, "shopping")`
2. 我们用 `KV.Registry.lookup(ets, "shopping")` 从表中获取信息
3. 命令返回 `:error`

4. 注册表进程创建了bucket，并且更新了缓存表

要修复这个问题，只需要让 `KV.Registry.create/2` 同步操作，使用 `call/2` 而不是 `cast/2` 。这就能保证客户端只会在表被修改后才能继续下面的操作。让我们来修改相应函数和回调：

```
def create(server, name) do
  GenServer.call(server, {:create, name})
end

def handle_call({:create, name}, _from, state) do
  case lookup(state.names, name) do
    {:ok, pid} ->
      {:reply, pid, state} # Reply with pid
    :error ->
      {:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
      ref = Process.monitor(pid)
      refs = HashDict.put(state.refs, ref, name)
      :ets.insert(state.names, {name, pid})
      GenEvent.sync_notify(state.events, {:create, name, pid})
      {:reply, pid, %{state | refs: refs}} # Reply with pid
  end
end
```

我们只是简单地把回调里的 `handle_cast/2` 改成了 `handle_call/3` ，并且返回创建的bucket的pid。

现在执行下测试。这次，我们要使用 `--trace` 选项：

```
$ mix test --trace
```

如果你的测试中有死锁或者竞争条件时， `--trace` 选项非常有用。因为它可以同步执行所有测试（而 `async: true` 没啥效果），并且显式每条测试的详细信息。这次我们应该只有一条失败（可能也是间歇性的）：

```
1) test removes buckets on exit (KV.RegistryTest)
   test/kv/registry_test.exs:48
   Assertion with == failed
   code: KV.Registry.lookup(ets, "shopping") == :error
   lhs:  {:ok, #PID<0.103.0>}
   rhs:  :error
   stacktrace:
     test/kv/registry_test.exs:52
```

根据错误信息，我们期望表中没有bucket，但是它却有。 这个问题和我们刚刚解决的相反：之前的问题是创建bucket的命令与更新表之间的延迟，而现在是bucket处理退出操作与清除它在表中的记录之间的延迟。

不幸的是，这次我们无法简单地把 `handle_info/2` 改成一个同步的操作。但是我们可以用事件管理器的通知来修复该失败。 先来看看我们 `handle_info/2` 的实现：

```
def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
  # 5. Delete from the ETS table instead of the HashDict
  {name, refs} = HashDict.pop(state.refs, ref)
  :ets.delete(state.names, name)
  GenEvent.sync_notify(state.event, {:exit, name, pid})
  {:noreply, %{state | refs: refs}}
end
```

注意我们在发通知之前就从ETS表中进行删除操作。这是有意为之的。 这意味着当我们收到 `{:exit, name, pid}` 通知的时候，表即已经是最新了。让我们更新剩下的代码：

```
test "removes buckets on exit", %{registry: registry, ets: ets} do
  KV.Registry.create(registry, "shopping")
  {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
  Agent.stop(bucket)
  assert_receive {:exit, "shopping", ^bucket} # Wait for event
  assert KV.Registry.lookup(ets, "shopping") == :error
end
```

我们对测试稍作调整，保证先收到 `{:exit, name, pid}消息，再执行` KV.Registry.lookup/2```。

你看，我们能够通过修改程序逻辑来使测试通过，而不是使用诸如 `:timer.sleep/1` 或者其它小技巧。这很重要。 大部分时间里，我们依赖于事件，监视以及消息机制来确保系统处在期望状态，在执行测试断言之前。

为方便，下面给出能通过的测试全文：

```
defmodule KV.RegistryTest do
  use ExUnit.Case, async: true

  defmodule Forwarder do
    use GenEvent

    def handle_event(event, parent) do
      send parent, event
      {:ok, parent}
    end
  end

  setup do
    {:ok, sup} = KV.Bucket.Supervisor.start_link
    {:ok, manager} = GenEvent.start_link
    {:ok, registry} = KV.Registry.start_link(:registry_table, manager, sup)

    GenEvent.add_mon_handler(manager, Forwarder, self())
    {:ok, registry: registry, ets: :registry_table}
  end

  test "sends events on create and crash", %{registry: registry, ets: ets} do
    KV.Registry.create(registry, "shopping")
    {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
    assert_receive {:create, "shopping", ^bucket}

    Agent.stop(bucket)
    assert_receive {:exit, "shopping", ^bucket}
  end

  test "spawns buckets", %{registry: registry, ets: ets} do
    assert KV.Registry.lookup(ets, "shopping") == :error

    KV.Registry.create(registry, "shopping")
    assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")

    KV.Bucket.put(bucket, "milk", 1)
    assert KV.Bucket.get(bucket, "milk") == 1
  end

  test "removes buckets on exit", %{registry: registry, ets: ets} do
    KV.Registry.create(registry, "shopping")
    {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
    Agent.stop(bucket)
    assert_receive {:exit, "shopping", ^bucket} # Wait for event
    assert KV.Registry.lookup(ets, "shopping") == :error
  end

  test "removes bucket on crash", %{registry: registry, ets: ets} do
    KV.Registry.create(registry, "shopping")
    {:ok, bucket} = KV.Registry.lookup(ets, "shopping")

    # Kill the bucket and wait for the notification
    Process.exit(bucket, :shutdown)
    assert_receive {:exit, "shopping", ^bucket}
    assert KV.Registry.lookup(ets, "shopping") == :error
  end
end
```

随着测试通过，我们只需更新监督者 `init/1` 回调函数的代码（文件 `lib/kv/supervisor.ex` ），传递ETS表的名字作为参数给注册表工人：

```
@manager_name KV.EventManager
@registry_name KV.Registry
@ets_registry_name KV.Registry
@bucket_sup_name KV.Bucket.Supervisor

def init(:ok) do
  children = [
    worker(GenEvent, [[name: @manager_name]]),
    supervisor(KV.Bucket.Supervisor, [[name: @bucket_sup_name]]),
    worker(KV.Registry, [@ets_registry_name, @manager_name,
                          @bucket_sup_name, [name: @registry_name]])
  ]

  supervise(children, strategy: :one_for_one)
end
```

注意我们仍使用`KV.Registry`作为ETS表的名字，好让debug方便些，因为它指明了使用它的模块。ETS名和进程名分别存储在不同的注册表，以避免冲突。

6.3-ETS当持久存储用

到目前为止，我们在初始化注册表的时候创建了一个ETS表，而没有操心在注册表结束时关闭该ETS表。 这是因为ETS表是“连接”（某种修辞上说）着创建它的进程的。如果那进程挂了，表也会自动关闭。

这作为默认行为实在是太方便了，我们可以在将来更多地利用这个特点。 记住，注册表和bucket监督者之间有依赖。注册表挂，我们希望bucket监督者也挂。 因为一旦注册表挂，所有连接bucket进程的信息都会丢失。 但是，假如我们能保存注册表的数据怎么样？ 如果我们能做到这点，就可以去除注册表和bucket监督者之间的依赖了，让`:one_for_one`成为监督者最合适的策略。

要做到这点需要些小改动。首先我们需要在监督者内启动ETS表。其次，我们需要把表的访问类型从`:protected`改成`:public`。 因为表的所有者是监督者，但是进行修改操作的仍然是时间管理者。

让我们从修改`KV.Supervisor`的`init/1`回调开始：

```
def init(:ok) do
  ets = :ets.new(@ets_registry_name,
                 [:set, :public, :named_table, {:read_concurrency, true}])

  children = [
    worker(GenEvent, [[name: @manager_name]]),
    supervisor(KV.Bucket.Supervisor, [[name: @bucket_sup_name]]),
    worker(KV.Registry, [ets, @manager_name,
                          @bucket_sup_name, [name: @registry_name]])
  ]

  supervise(children, strategy: :one_for_one)
end
```

接下来，我们修改`KV.Registry`的`init/1`回调，因为它不再需要创建一个表，而是需要一个表作为参数：

```
def init({table, events, buckets}) do
  refs = HashDict.new
  {:ok, %{names: table, refs: refs, events: events, buckets: buckets}}
end
```

最终，我们修改`test/kv/registry_test.exs`中的`setup`回调，来显式地创建ETS表。 我们还将用这个机会分离`setup`的功能，放到一个方便的私有函数中：

```
setup do
  ets = :ets.new(:registry_table, [:set, :public])
  registry = start_registry(ets)
  {:ok, registry: registry, ets: ets}
end

defp start_registry(ets) do
  {:ok, sup} = KV.Bucket.Supervisor.start_link
  {:ok, manager} = GenEvent.start_link
  {:ok, registry} = KV.Registry.start_link(ets, manager, sup)

  GenEvent.add_mon_handler(manager, Forwarder, self())
  registry
end
```

这之后，我们的测试应该都绿啦！

现在只剩下一个场景需要考虑：一旦我们收到了ETS表，可能有现存的bucket的pid在这个表中。 这是我们这次改动的目的。 但是，新启动的注册表进程没有监视这些bucket，因为它们是作为之前的注册表的一部分创建的，现在那些注册表已经不存在了。 这意味着表将被严重拖累，因为我们都不去清除已经挂掉的bucket。

来增加一个测试来暴露这个bug：

```
test "monitors existing entries", %{registry: registry, ets: ets} do
  bucket = KV.Registry.create(registry, "shopping")

  # Kill the registry. We unlink first, otherwise it will kill the test
  Process.unlink(registry)
  Process.exit(registry, :shutdown)

  # Start a new registry with the existing table and access the bucket
  start_registry(ets)
  assert KV.Registry.lookup(ets, "shopping") == {:ok, bucket}

  # Once the bucket dies, we should receive notifications
  Process.exit(bucket, :shutdown)
  assert_receive {:exit, "shopping", ^bucket}
  assert KV.Registry.lookup(ets, "shopping") == :error
end
```

执行这个测试，它将失败：

```
1) test monitors existing entries (KV.RegistryTest)
   test/kv/registry_test.exs:72
   No message matching {:exit, "shopping", ^bucket}
   stacktrace:
     test/kv/registry_test.exs:85
```

这是我们期望的。如果bucket不被监视，在它挂的时候，注册表将得不到通知，因此也没有事件发生。 我们可以修改 `KV.Registry` 的 `init/1` 回调来修复这个问题。给所有表中的现存条目设置监视器：

```
def init({table, events, buckets}) do
  refs = :ets.foldl(fn {name, pid}, acc ->
    HashDict.put(acc, Process.monitor(pid), name)
  end, HashDict.new, table)

  {:ok, %{names: table, refs: refs, events: events, buckets: buckets}}
end
```

我们用 `:ets.foldl/3` 来遍历表中所有条目，类似于 `Enum.reduce/3` 。它为每个条目执行提供的函数，并且用一个累加器累加结果。 在函数回调中，我们监视每个表中的pid，并相应地更新存放引用信息的字典。 如果有某个条目是挂掉的，我们还能收到 `:DOWN` 消息，稍后可以清除它们。

本章让监督者拥有ETS表，并且使其将表作为参数传递给注册表进程。通过这样的方法，我们让程序变得更加健壮。 我们还探索了把ETS当作缓存，并且讨论了如果在客户端和服务器共享数据时会进入的竞争状态。

7-依赖和伞工程

本章我们简短地讨论在Mix中如何管理依赖。

我们的应用程序 `kv` 完成了，现在是时候实现我们第一章提到的那个处理请求的服务器了：

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

我们不会再往 `kv` 应用程序中添加代码，而是来创建一个TCP服务器作为 `kv` 的客户端。 因为整个Elixir生态系统被设定为更加适应应用程序。因此我们最好是把工程分成小的应用程序，而不是大一统。

在创建新的应用程序之前，我们必须先来讨论Mix如何处理依赖。 实际中，我们会遇到两种依赖：内部的和外部的。Mix都支持。

7.1-外部依赖

外部的依赖不是你整的。比如，你要发布的KV程序，它需要HTTP的API，你可以用Plug作为一个外部依赖。

安装外部依赖很简单。通常我们使用Hex包管理器。你只要在 `mix.exs` 中的deps函数中列出依赖：

```
def deps do
  [{:plug, "~> 0.5.0"}]
end
```

这个依赖引用的是plug在0.5.x系列版本中最新推送给Hex的一个。`~>`后面跟着版本数字。更多关于如何指定某特定版本的信息，请参考Version模块文档。

通常推给Hex的都是稳定的发行版。如果你想依赖的是正在开发中的最新版本，那么Mix还支持你引用git中的相应repo：

```
def deps do
  [{:plug, git: "git://github.com/elixir-lang/plug.git"}]
end
```

你会注意到，当你给你的工程加了一个依赖后，Mix会生成了一个`mix.lock`文件，用该文件确保可重复的构建。 lock文件必须ckeckin到你的版本管理系统中去，来保证任何使用这个工程的人都拥有和你同样的依赖的版本。

Mix提供了很多依赖相关的操作，可以用`mix help`查看：

```
$ mix help
mix deps              # List dependencies and their status
mix deps.clean        # Remove the given dependencies' files
mix deps.compile      # Compile dependencies
mix deps.get          # Get all out of date dependencies
mix deps.unlock       # Unlock the given dependencies
mix deps.update       # Update the given dependencies
```

最常用的操作是`mix deps.get`和`mix deps.update`。 一旦获取了依赖程序，这些程序会自动被编译好供你使用。 关于deps操作，可以通过`mix help deps`或者Mix.Tasks.Deps模块文档阅读更多信息。

7.2-内部依赖

内部的依赖指的是你在同一个工程中的某个程序。

8-Task模块和通用TCP服务器（gen_tcp）

本章我们学习如何使用Erlang的:gen_tcp模块来处理请求。 在未来几章中我们还会扩展我们的服务器，使之能够服务于真正的命令。 这也是我们探索Elixir的Task模块的大好机会。

8.1-Echo服务器

我们首先实现一个Echo（回声）服务器来开始我们的TCP服务器之旅。它只是简单地返回从请求中收到的文字。 我们会慢慢地改进这个服务器，使它有监督者来监督，并且可以处理大量连接。

一个TCP服务器，总的来说，实现以下几步：

1. 在可用端口建立socket连接，监听这个端口
2. 等待这个端口的客户端连接，有了就接受它
3. 读取客户端请求并且写回复

我们来实现这些步骤。在 `apps/kv_server` 程序中，打开文件 `lib/kv_server.ex` ，添加以下函数：

```
def accept(port) do
  # The options below mean:
  #
  # 1. `:binary` - receives data as binaries (instead of lists)
  # 2. `packet: :line` - receives data line by line
  # 3. `active: false` - block on `:gen_tcp.recv/2` until data is available
  #
  {:ok, socket} = :gen_tcp.listen(port,
                    [:binary, packet: :line, active: false])
  IO.puts "Accepting connections on port #{port}"
  loop_acceptor(socket)
end

defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  serve(client)
  loop_acceptor(socket)
end

defp serve(client) do
  client
  |> read_line()
  |> write_line(client)

  serve(client)
end

defp read_line(socket) do
  {:ok, data} = :gen_tcp.recv(socket, 0)
  data
end

defp write_line(line, socket) do
  :gen_tcp.send(socket, line)
end
```

我们通过调用 `KVServer.accept(4040)` 来启动服务器，其中4040是端口号。在 `accept/1` 中，第一步是去监听这个端口，知道socket变成可用状态，然后调用 `loop_acceptor/1` 。函数 `loop_acceptor/1` 只是一个循环，来接受客户端的连接。 对于每次接受的客户端连接，我们调用 `serve/1` 函数。

函数 `serve/1` 也是个循环，它一次从socket中读取一行，并将其写进给socket的回复。 注意 `serve/1` 使用了管道运算符 `|>` 来表达操作流程。 管道运算符计算左侧函数计算的结果，并将其作为第一个参数传递给右侧函数调用。如：

```
socket |> read_line() |> write_line(socket)
```

相当于：

```
write_line(read_line(socket), socket)
```

> 当使用 `|>` 运算符时，是否给函数调用加上括号是很重要的。举个例子：
>
> ```
> 1..10 |> Enum.filter &(&1 <= 5) |> Enum.map &(&1 * 2)
> ```
>
> 会被翻译为：
>
> ```
> 1..10 |> Enum.filter(&(&1 <= 5) |> Enum.map(&(&1 * 2)))
> ```
>
> 这个不是我们想要的，因为本应传给 `Enum.filter/2` 的那个匿名函数 `&(&1<=5)` 成了传给 `Enum.map/2` 的第一个参数。 解决方法就是加上括号：
>
> ```
> 1..10 |> Enum.filter(&(&1 <= 5)) |> Enum.map(&(&1 * 2))
> ```

函数 `read_line/2` 中使用 `:gen_tcp.recv/2` 接收从socket传来的数据。 而 `write_line/2` 中使用 `:gen_tcp.send/2` 向socket写入数据。

这差不多就是我们为实现这个回声服务器所要做的。让我们试一试。

用 `iex -S mix` 在 `kv_server` 应用程序中启动对话，执行：

```
iex> KVServer.accept(4040)
```

服务器就运行了，注意到此时该命令行会被阻塞。现在我们使用一个telnet客户端 来访问这个服务器。基本上每个操作系统都有telnet客户端程序，命令也都差不多：

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello
is it me
is it me
you are looking for?
you are looking for?
```

输入“hello”，按回车，你就会得到“hello”字样的回复。好牛逼！

退出telnet客户端方法不一，有些用 `ctrl + ]` ，有些是 `quit` 按回车。

一旦你退出telnet客户端，你会发现IEx会话中打印出一个错误信息：

```
** (MatchError) no match of right hand side value: {:error, :closed}
    (kv_server) lib/kv_server.ex:41: KVServer.read_line/1
    (kv_server) lib/kv_server.ex:33: KVServer.serve/1
    (kv_server) lib/kv_server.ex:27: KVServer.loop_acceptor/1
```

这是因为我们还期望从 `:gen_tcp.recv/2` 拿数据，但是客户端断了。我们将来要处理这个问题才行。

目前还有个更重要的bug要修：假如TCP接收者挂了怎么办？意为它没有监督者，不会自己重启，要是挂了我们将不能在处理更多的请求。 这就是为啥我们要将它挪进监督树。

8.2-Tasks

我们已经学习了Agent，通用服务器以及事件管理器。它们都可以进行多消息协作，或者管理状态。 但是，若是只需要处理一些任务，选什么呢？

Task模块为此提供了所需的功能。 例如，它有 `start_link/3` 函数，接受一个模块名、一个函数和函数的参数，从而执行这个传入的函数，并且还是作为监督树的一部分。

我们来试试。打开 `lib/kv_server.ex` ，修改下里 `start/2` 函数里的监督者：

```
def start(_type, _args) do
  import Supervisor.Spec

  children = [
    worker(Task, [KVServer, :accept, [4040]])
  ]

  opts = [strategy: :one_for_one, name: KVServer.Supervisor]
  Supervisor.start_link(children, opts)
end
```

改动的意思是要让 `KVServer.accept(4040)` 成为一个工人来运行。目前我们暂时hardcode这个端口号，之后再讨论如何修改。

现在，这个服务器是监督树的一部分了，它应该会随着应用程序启动而自动运行。 在终端中输入 `mix run --no-halt` ，然后再次用telnet客户端来试试看是否还一切正常：

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
say you
say you
say me
say me
```

看，它还是好使！这回就算退了客户端，服务器挂了，你会看到又一个立马起来了。嗯，不错。。。不过它可伸缩性如何？

试着打开两个telnet客户端一起连接，你会注意到，第二个客户端根本不能回声：

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello?
HELLOOOOOO?
```

看起来根本不工作嘛。这是因为处理请求和接受请求是在同一个进程。一个客户端连上来，就没法处理第二个了。

8.3-Task的监督者

为了让我们的服务器能够处理并发连接，我们需要让一个进程来当接收者，然后派生其它的进程来服务接收到的连接。 一个方案是：

```
defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  serve(client)
  loop_acceptor(socket)
end
```

函数 `Task.start_link/1` 类似 `Task.start_link/3` ，但是它可以接受一个匿名函数而不是（模块，函数，参数）的组合：

```
defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  Task.start_link(fn -> serve(client) end)
  loop_acceptor(socket)
end
```

我们翻过这个错了，记得吗？

和我们当时在注册表进程中调用 `KV.Bucket.start_link/0` 犯的错差不多。它意味着一个bucket挂会导致整个注册表进程挂。

上面的代码页犯了相同的错误：如果我们把 `serve(client)` 这个任务和接收者连接起来，那么在处理请求时发生的小事故就会导致请求接收者挂，继而导致连接都挂掉。

当时我们解决这个问题是用了一个简单的一对一监督者。这里我们也将使用相同的办法，除了一点：这个模式在Task中实在是太通用了， 所有Task已经为之提供了一个解决方案---一个简单的一对一监督者加上临时工（临时的工人），这个我们在之前的监督树中就是这么用的。

让我们再次修改下 `start/2` 函数，加个监督者：

```
def start(_type, _args) do
  import Supervisor.Spec

  children = [
    supervisor(Task.Supervisor, [[name: KVServer.TaskSupervisor]]),
    worker(Task, [KVServer, :accept, [4040]])
  ]

  opts = [strategy: :one_for_one, name: KVServer.Supervisor]
  Supervisor.start_link(children, opts)
end
```

我们简单地启动了一个 `Task.Supervisor` 进程，名字叫 `Task.Supervisor` 。记住，因为接收者任务依赖于这个监督者，因此该监督者必须先启动。

现在我们只需修改 `loop_acceptor/2` ，使用 `Task.Supervisor` 来处理每个请求：

```
defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  Task.Supervisor.start_child(KVServer.TaskSupervisor, fn -> serve(client) end)
  loop_acceptor(socket)
end
```

用命令 `mix run --no-halt` 启动新的服务器，现在就可以打开多个客户端来连接了。而且你会发现一个客户端退出不会让接收者挂掉。 好棒！

一下是完整的服务器实现，在单个模块中：

```
defmodule KVServer do
  use Application

  @doc false
  def start(_type, _args) do
    import Supervisor.Spec

    children = [
      supervisor(Task.Supervisor, [[name: KVServer.TaskSupervisor]]),
      worker(Task, [KVServer, :accept, [4040]])
    ]

    opts = [strategy: :one_for_one, name: KVServer.Supervisor]
    Supervisor.start_link(children, opts)
  end

  @doc """
  Starts accepting connections on the given `port`.
  """
  def accept(port) do
    {:ok, socket} = :gen_tcp.listen(port,
                      [:binary, packet: :line, active: false])
    IO.puts "Accepting connections on port #{port}"
    loop_acceptor(socket)
  end

  defp loop_acceptor(socket) do
    {:ok, client} = :gen_tcp.accept(socket)
    Task.Supervisor.start_child(KVServer.TaskSupervisor, fn -> serve(client) end)
    loop_acceptor(socket)
  end

  defp serve(socket) do
    socket
    |> read_line()
    |> write_line(socket)

    serve(socket)
  end

  defp read_line(socket) do
    {:ok, data} = :gen_tcp.recv(socket, 0)
    data
  end

  defp write_line(line, socket) do
    :gen_tcp.send(socket, line)
  end
end
```

因为我们修改了监督者的需求，我们会问：我们的监督者策略还适用吗？

这里答案是Yes：如果接收者挂了，现存的连接是没理由一起挂的。另一方面，如果task监督者挂了，同样也没必要让接收者挂掉。 这和注册表进程那种情况相反，那种情况我们在一开始必须在注册表进程挂掉时让监督者也挂掉，直到后来我们用上了ETS来持久化保存状态。而task是没有状态什么的，挂掉一个两个也不会拖谁的后腿。

下一章我们将开始解析客户请求，然后发送回复，从而完成我们的服务器。

9-文档、测试和管道

本章我们将实现“解析”功能，来解析在第一章提到的命令行操作指令（还记得吗？我们在写一个简易redis！）：

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

解析功能完成后，我们会把代码更新到之前创建的 `:kv` 程序里面去。

文档测试（Doctests）

> Doctest常见于python，ruby等语言，是一种基于代码注释文档书写单元测试的方法。 用特定的语法为函数或方法书写注释，用doctest命令执行这些文档测试代码。
> 注意：为了保证代码简洁，不能完全用doctest代替传统的单元测试代码

在语言官网首页，我们说Elixir视代码中的文档标记为语言中的一等公民。在文档手册中，我们也多次涉及这个概念。 比如经常在IEx命令行中执行 `mix help` ，以及输入 `h Enum` 或 `h` + 其他模块名字。

本章中，我们在实现上文所说“解析”功能的时候，引入文档注释的内容。它能够让我们直接通过代码的文档注释写测试， 有助于我们在文档注释中写出更准确的sample。

现在来创建我们的解析功能函数 `lib/kv_server/command.ex` 。先写doctest：

```
defmodule KVServer.Command do
  @doc ~S"""
  Parses the given `line` into a command.

  ## Examples

      iex> KVServer.Command.parse "CREATE shopping\r\n"
      {:ok, {:create, "shopping"}}

  """
  def parse(line) do
    :not_implemented
  end
end
```

Doctests规定的书写形式：在4空格缩进之后的 `iex>` 提示符后。如果一个命令不止一行，则在除第一行的其它行用 `...>` 代替 `iex>` 字样。命令的结果则写在 `iex>` 或 `...>` 的下一行，后面以一个新行或者下一个新的 `iex>` 行结束。

同时注意，我们写注释文档时用 `@doc ~S"""` 起头。 `~S` 可以保证文档里写的 `/r/n` 不会在执行doctests测试前被转义成回车。

执行doctets，我们先创建一个测试脚本 `test/kv_server/command_test.exs` ，在用例中调用 `doctest KVServer.Command` ：

```
defmodule KVServer.CommandTest do
  use ExUnit.Case, async: true
  doctest KVServer.Command
end
```

10-分布式任务及配置

在这最后一章中，我们将回到 `:kv` 应用程序，给它添加一个路由层，使之可以根据桶的名字，在各个节点间分发请求。
路由层会接收一个如下形式的路由表：

```
[{?a..?m, :"foo@computer-name"},
 {?n..?z, :"bar@computer-name"}]
```

路由者（负责转发请求的角色，可能是个节点）将根据桶名字的第一个字节查这个路由表，然后根据路由表所示将用户对桶的请求发给相应的节点。比如，根据上表， 某个桶名字第一个字母是“a”（ `?a` 表示字母“a”的Unicode码），那么对它的请求会被路由到 `foo@computer-name` 这个节点去。

如果节点处理了路由请求，那么路由过程结束。如果节点自己也有个路由表，会根据该路由表把请求又转发给了其它相应节点。 如果最终没有节点接收和处理请求，则报出异常。

你会问，为啥我们不简单地命令路由表中查出来的节点来直接处理数据请求，而是将路由请求传给它？ 当一个路由表像上面那个例子一样简单的话，这样做比较简单。但是，考虑到程序的规模越来越大，会将大路由表分解成小块， 分开存放在不同节点。这种情况下，还是用分发路由请求的方式简单些。也许在某些时刻， 节点 `foo@computer-name` 将只负责路由请求，不再处理桶数据请求，它承载的桶都会分给其它节点。 这种方式，节点 `bar@computer-name` 都不需要知道这个变化。

> 注意： 本章中我们会在同一台机器上使用两个节点。你当然可以用同一网络中得不同的机器，但是这种方式需要做一些准备。 首先你需要确保所有的机器上都有一个名叫 `~/.erlang.cookie` 的文件。 其次，你要保证 epmd 在一个可访问的端口运行（你可以执行 `epmd -d` 查看debug信息来确定这点）。 第三，如果你想学习更多关于分布式编程的知识，我们推荐这篇文章。

我们最初版本的分布式代码

Elixir内置了连接节点及于期间交换信息的工具。事实上，在一个分布式的环境中进行的消息的发送和接收， 和之前学习的进程的内容并无区别：因为Elixir的进程是 位置透明 的。 意思是当我们发送消息的时候， 它不管请求是在当前节点还是在别的节点，虚拟机都会传递消息。

为了执行分布式的代码，我们需要用某个名字启动虚拟机。 名字可以使简短的（当在同一个网络内）或是较长的（需要附上计算机地址）。让我们启动一个新IEx会话：

```
$ iex --sname foo
```

You can see now the prompt is slightly different and shows the node name followed by the computer name:

```
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)
iex(foo@jv)1>
```

My computer is named `jv` , so I see `foo@jv` in the example above, but you will get a different result. We will use `jv@computer-name` in the following examples and you should update them accordingly when trying out the code.

让我们在shell中定义一个名叫 `Hello` 的模块：

```
iex> defmodule Hello do
...>  def world, do: IO.puts "hello world"
...> end
```

If you have another computer on the same network with both Erlang and Elixir installed, you can start another shell on it. If you don't, you can simply start another IEx session in another terminal. In either case, give it the short name of `bar` :

```
$ iex --sname bar
```

Note that inside this new IEx session, we cannot access `Hello.world/0` :

```
iex> Hello.world
** (UndefinedFunctionError) undefined function: Hello.world/0
    Hello.world()
```

However we can spawn a new process on `foo@computer-name` from `bar@computer-name` ! Let's give it a try (where `@computer-name` is the one you see locally):

```
iex> Node.spawn_link :"foo@computer-name", fn -> Hello.world end
#PID<9014.59.0>
hello world
```

Elixir spawned a process on another node and returned its pid. The code then executed on the other node where the `Hello.world/0` function exists and invoked that function. Note that the result of "hello world" was printed on the current node `bar` and not on `foo` . In other words, the message to be printed was sent back from `foo` to `bar` . This happens because the process spawned on the other node (`foo`) still has the group leader of the current node (`bar`). We have briefly talked about group leaders in the IO chapter.

We can send and receive message from the pid returned by `Node.spawn_link/2` as usual. Let's try a quick ping-pong example:

```
iex> pid = Node.spawn_link :"foo@computer-name", fn ->
...>   receive do
...>     {:ping, client} -> send client, :pong
...>   end
...> end
#PID<9014.59.0>
iex> send pid, {:ping, self}
{:ping, #PID<0.73.0>}
iex> flush
:pong
:ok
```

From our quick exploration, we could conclude that we should simply use `Node.spawn_link/2` to spawn processes on a remote node every time we need to do a distributed computation. However we have learned throughout this guide that spawning processes outside of supervision trees should be avoided if possible, so we need to look for other options.

There are three better alternatives to `Node.spawn_link/2` that we could use in our implementation:

1. We could use Erlang's :rpc module to execute functions on a remote node. Inside the `bar@computer-name` shell above, you can call `:rpc.call(:"foo@computer-name", Hello, :world, [])` and it will print "hello world"
2. We could have a server running on the other node and send requests to that node via the GenServer API. For example, you can call a remote named server using `GenServer.call({name, node}, arg)` or simply passing the remote process PID as first argument
3. We could use tasks, which we have learned about in a previous chapter, as they can be spawned on both local and remote nodes

The options above have different properties. Both `:rpc` and using a GenServer would serialize your requests on a single server, while tasks are effectively running asynchronously on the remote node, with the only serialization point being the spawning done by the supervisor.

For our routing layer, we are going to use tasks, but feel free to explore the other alternatives too.

async/await

So far we have explored tasks that are started and run in isolation, with no regard for their return value. However, sometimes it is useful to run a task to compute a value and read its result later on. For this, tasks also provide the `async/await` pattern:

```
task = Task.async(fn -> compute_something_expensive end)
res  = compute_something_else()
res + Task.await(task)
```

`async/await` provides a very simple mechanism to compute values concurrently. Not only that, `async/await` can also be used with the same `Task.Supervisor` we have used in previous chapters. We just need to call `Task.Supervisor.async/2` instead of `Task.Supervisor.start_child/2` and use `Task.await/2` to read the result later on.

Distributed tasks

Distributed tasks are exactly the same as supervised tasks. The only difference is that we pass the node name when spawning the task on the supervisor. Open up `lib/kv/supervisor.ex` from the `:kv` application. Let's add a task supervisor to the tree:

```
supervisor(Task.Supervisor, [[name: KV.RouterTasks]]),
```

Now, let's start two named nodes again, but inside the `:kv` application:

```
$ iex --sname foo -S mix
$ iex --sname bar -S mix
```

From inside `bar@computer-name`, we can now spawn a task directly on the other node via the supervisor:

```
iex> task = Task.Supervisor.async {KV.RouterTasks, :"foo@computer-name"}, fn ->
...>   {:ok, node()}
...> end
%Task{pid: #PID<12467.88.0>, ref: #Reference<0.0.0.400>}
iex> Task.await(task)
{:ok, :"foo@computer-name"}
```

Our first distributed task is straightforward: it simply gets the name of the node the task is running on. With this knowledge in hand, let's finally write the routing code.

路由层

Create a file at `lib/kv/router.ex` with the following contents:

```
defmodule KV.Router do
  @doc """
  Dispatch the given `mod`, `fun`, `args` request
  to the appropriate node based on the `bucket`.
  """
  def route(bucket, mod, fun, args) do
    # Get the first byte of the binary
    first = :binary.first(bucket)

    # Try to find an entry in the table or raise
    entry =
      Enum.find(table, fn {enum, node} ->
        first in enum
      end) || no_entry_error(bucket)

    # If the entry node is the current node
    if elem(entry, 1) == node() do
      apply(mod, fun, args)
    else
      sup = {KV.RouterTasks, elem(entry, 1)}
      Task.Supervisor.async(sup, fn ->
        KV.Router.route(bucket, mod, fun, args)
      end) |> Task.await()
    end
  end

  defp no_entry_error(bucket) do
    raise "could not find entry for #{inspect bucket} in table #{inspect table}"
  end

  @doc """
  The routing table.
  """
  def table do
    # Replace computer-name with your local machine name.
    [{?a..?m, :"foo@computer-name"},
     {?n..?z, :"bar@computer-name"}]
  end
end
```

Let's write a test to verify our router works. Create a file named `test/kv/router_test.exs` containing:

```
defmodule KV.RouterTest do
  use ExUnit.Case, async: true

  test "route requests across nodes" do
    assert KV.Router.route("hello", Kernel, :node, []) ==
           :"foo@computer-name"
    assert KV.Router.route("world", Kernel, :node, []) ==
           :"bar@computer-name"
  end

  test "raises on unknown entries" do
    assert_raise RuntimeError, ~r/could not find entry/, fn ->
      KV.Router.route(<<0>>, Kernel, :node, [])
    end
  end
end
```

The first test simply invokes `Kernel.node/0`, which returns the name of the current node, based on the bucket names "hello" and "world". According to our routing table so far, we should get `foo@computer-name` and `bar@computer-name` as responses, respectively.

The second test just checks that the code raises for unknown entries.

In order to run the first test, we need to have two nodes running. Let's restart the node named `bar`, which is going to be used by tests. This time we'll need to run the node in the `test` environment, to ensure the compiled code being run is exactly the same as that used in the tests themselves:

```
$ MIX_ENV=test iex --sname bar -S mix
```

And now run tests with:

```
$ elixir --sname foo -S mix test
```

Our test should successfully pass. Excellent!

Test filters and tags

Although our tests pass, our testing structure is getting more complex. In particular, running tests with only `mix test` causes failures in our suite, since our test requires a connection to another node.

Luckily, ExUnit ships with a facility to tag tests, allowing us to run specific callbacks or even filter tests altogether based on those tags.

All we need to do to tag a test is simply call `@tag` before the test name. Back to `test/kv/router_test.exs`, let's add a `:distributed` tag:

```
@tag :distributed
test "route requests across nodes" do
```

Writing `@tag :distributed` is equivalent to writing `@tag distributed: true`.

With the test properly tagged, we can now check if the node is alive on the network and, if not, we can exclude all distributed tests. Open up `test/test_helper.exs` inside the `:kv` application and add the following:

```
exclude =
  if Node.alive?, do: [], else: [distributed: true]

ExUnit.start(exclude: exclude)
```

Now run tests with `mix test`:

```
$ mix test
Excluding tags: [distributed: true]

.......

Finished in 0.1 seconds (0.1s on load, 0.01s on tests)
7 tests, 0 failures
```

This time all tests passed and ExUnit warned us that distributed tests were being excluded. If you run tests with `$ elixir --sname foo -S mix test`, one extra test should run and successfully pass as long as the `bar@computer-name` node is available.

The `mix test` command also allows us to dynamically include and exclude tags. For example, we can run `$ mix test --include distributed` to run distributed tests regardless of the value set in `test/test_helper.exs`. We could also pass `--exclude` to exclude a particular tag from the command line. Finally, `--only` can be used to run only tests with a particular tag:

```
$ elixir --sname foo -S mix test --only distributed
```

You can read more about filters, tags and the default tags in `ExUnit.Case` module documentation.

Application environment and configuration

So far we have hardcoded the routing table into the `KV.Router` module. However, we would like to make the table dynamic. This allows us not only to configure development/test/production, but also to allow different nodes to run with different entries in the routing table. There is a feature of OTP that does exactly that: the application environment.

Each application has an environment that stores the application's specific configuration by key. For example, we could store the routing table in the `:kv` application environment, giving it a default value and allowing other applications to change the table as needed.

Open up `apps/kv/mix.exs` and change the `application/0` function to return the following:

```
def application do
  [applications: [],
   env: [routing_table: []],
   mod: {KV, []}]
end
```

We have added a new `:env` key to the application. It returns the application default environment, which has an entry of key `:routing_table` and value of an empty list. It makes sense for the application environment to ship with an empty table, as the specific routing table depends on the testing/deployment structure.

In order to use the application environment in our code, we just need to replace `KV.Router.table/0` with the definition below:

```
@doc """
The routing table.
"""
def table do
  Application.get_env(:kv, :routing_table)
end
```

We use `Application.get_env/2` to read the entry for `:routing_table` in `:kv` 's environment. You can find more information and other functions to manipulate the app environment in the Application module.

Since our routing table is now empty, our distributed test should fail. Restart the apps and re-run tests to see the failure:

```
$ iex --sname bar -S mix
$ elixir --sname foo -S mix test --only distributed
```

The interesting thing about the application environment is that it can be configured not only for the current application, but for all applications. Such configuration is done by the `config/config.exs` file. For example, we can configure IEx default prompt to another value. Just open `apps/kv/config/config.exs` and add the following to the end:

```
config :iex, default_prompt: ">>>"
```

Start IEx with `iex -S mix` and you can see that the IEx prompt has changed.

This means we can configure our `:routing_table` directly in the `config/config.exs` file as well:

```
# Replace computer-name with your local machine nodes.
config :kv, :routing_table,
       [{?a..?m, :"foo@computer-name"},
        {?n..?z, :"bar@computer-name"}]
```

Restart the nodes and run distributed tests again. Now they should all pass.

Each application has its own `config/config.exs` file and they are not shared in any way. Configuration can also be set per environment. Read the contents of the config file for the `:kv` application for more information on how to do so.

Since config files are not shared, if you run tests from the umbrella root, they will fail because the configuration we just added to `:kv` is not available there. However, if you open up `config/config.exs` in the umbrella, it has instructions on how to import config files from children applications. You just need to invoke:

```
import_config "../apps/kv/config/config.exs"
```

The `mix run` command also accepts a `--config` flag, which allows configuration files to be given on demand. This could be used to start different nodes, each with its own specific configuration (for example, different routing tables).

Overall, the built-in ability to configure applications and the fact that we have built our software as an umbrella application gives us plenty of options when deploying the software. We can:

- deploy the umbrella application to a node that will work as both TCP server and key-value storage
- deploy the `:kv_server` application to work only as a TCP server as long as the routing table points only to other nodes
- deploy only the `:kv` application when we want a node to work only as storage (no TCP access)

As we add more applications in the future, we can continue controlling our deploy with the same level of granularity, cherry-picking which applications with which configuration are going to production. We can also consider building multiple releases with a tool like exrm, which will package the chosen applications and configuration, including the current Erlang and Elixir installations, so we can deploy the application even if the runtime is not pre-installed on the target system.

Finally, we have learned some new things in this chapter, and they could be applied to the `:kv_server` application as well. We are going to leave the next steps as an exercise:

- change the `:kv_server` application to read the port from its application environment instead of using the hardcoded value of 4040
- change and configure the `:kv_server` application to use the routing functionality instead of dispatching directly to the local `KV.Registry`. For `:kv_server` tests, you can make the routing table simply point to the current node itself

总结

这一章我们创建了一个简单的路由，并通过它探索了Elixir以及Erlang虚拟机的分布式特性，学习了如何配置路由表。这是《Elixir高级编程手册（Mix和OTP）》的最后一章。

通过这本手册，我们编写了一个非常简单的分布式键-值存储程序，领略了许多重要概念，如通用服务器、事件管理者、监督者、任务、代理、应用程序等等。不仅如此，我们还为整个程序写了测试代码，熟悉了ExUnit，还学习了如何使用Mix构建工具来完成许许多多的工作。

如果你要找一个生成环境能用的分布式键-值存储，你一定要去看看Riak，它也运行于Erlang VM之上。Riak中，桶是有冗余的，以防止数据丢失。另外，它用相容哈希（consistent hashing）而不是路由机制来匹配桶和节点。因为相容哈希算法可以减少因为桶名冲突而导致的不得不将桶迁移到新节点的开销。